NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

AUTOMATED NETWORK PROTOCOL REACHABILITY ANALYSIS WITH SUPERTRACE ALGORITHM AND TESTGEN : AUTOMATED GENERATION OF TEST SEQUENCE FOR A FORMAL PROTOCOL SPECIFICATION

by

Cuneyt BASARAN
March 1994

Thesis Advisor: Gilbert M. Lundy

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1994	Master's Thesis

4. TITLE AND SUBTITLE
Automated Network Protocol Reachability Analysis with Supertrace Algorithm and TESTGEN: Automated Generation of Test Sequence for a Formal Protocol Specification.

5. FUNDING NUMBERS

6. AUTHOR(S)
Basaran, Cuneyt

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The automation of reachability analysis is an important step in verification of network protocols. The memory size needed for the full state analysis of complex protocols is usually very large and not available on most of the systems. A controlled partial search algorithm "Supertrace" is implemented in this thesis to analyze protocols that can not be analyzed efficiently by full state search method. Supertrace algorithm provided the analysis of large protocols by generating 80% to 95% more states and is much faster as total process time than full state analysis.

Second problem addressed in this thesis is the improvement of conformance testing for protocol implementations. The "conformance testing" is used to check that the external behavior of a given implementation of a protocol is equivalent to its formal specification. A previously created procedure for conformance test sequence generation is automated in this thesis by the ADA programming language. The software tool implemented, uses a protocol specified formally with *systems of communicating machines* and creates test sequences as output. The tool was applied to a formal specification of the CSMA/CD and FDDI protocols and .the results obtained, was consistent with the previous results. The automation of the tool expanded the applicability of the previous procedure to larger and more complex protocols.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Supertrace, Network Protocols, Reachability Analysis, Conformance Test, System of Communicating Machines Protocol Model	102
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

Approved for public release; distribution is unlimited

AUTOMATED NETWORK PROTOCOL REACHABILITY ANALYSIS WITH SUPERTRACE ALGORITHM AND TESTGEN : AUTOMATED GENERATION OF TEST SEQUENCE FOR A FORMAL PROTOCOL SPECIFICATION

by
Cuneyt BASARAN
LTJG. Turkish Navy
BS, Turkish Naval Academy ,1988

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1994

Author: _____
Cuneyt Basaran

Approved By: _____
Gilbert M. Lundy, Thesis Advisor

_____
Lou Stevens, Second Reader

_____
Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

The automation of reachability analysis is an important step in verification of network protocols. The memory size needed for the full state analysis of complex protocols is usually very large and not available on most of the systems. A controlled partial search algorithm "Supertrace" is implemented in this thesis to analyze protocols that can not be analyzed efficiently by full state search method. Supertrace algorithm provided the analysis of large protocols by generating 80% to 95% more states and is much faster as total process time than full state analysis.

Second problem addressed in this thesis is the improvement of conformance testing for protocol implementations. The "conformance testing" is used to check that the external behavior of a given implementation of a protocol is equivalent to its formal specification. A previously created procedure for conformance test sequence generation is automated in this thesis by the ADA programming language. The software tool implemented, uses a protocol specified formally with *systems of communicating machines* and creates test sequences as output. The tool was applied to a formal specification of the CSMA/CD and FDDI protocols and the results obtained, was consistent with the previous results. The automation of the tool expanded the applicability of the previous procedure to larger and more complex protocols.

ACKNOWLEDGEMENTS

I would like to thank my advisor Prof. G. M. Lundy, who provided me a great deal of guidance in reachability analysis of network protocols and conformance testing of protocol implementations on which this thesis was produced. His continuous support, enthusiasm, and patience were invaluable assets for the completion of this work.

I also would like to thank Prof. Lou Stevens, for his interest and support on the area of my thesis work.

DEDICATION

I dedicate my thesis to my dear wife Emel and our daughter Ipek Ece who were my inspiration for completing my degree and kept me looking to the future. They were very supportive throughout my study program and gave their continuous patience and love.

TABLE OF CONTENTS

I. INTRODUCTION

A. Background

Systems of communicating machines (SCM) [LUND88] is a formal protocol model introduced during the last decade, which is used for specification, verification and analysis of communication protocols. The main goal of the SCM model was to improve the well-known simpler Communicating Finite State Machines (CFSM) model. In several papers the model was used to specify and verify several communication protocols. The analysis which is carried out with the model, called system state analysis, has been automated. The SCM model of a protocol can then be easily verified.

This model uses a combination of finite state machines and variables. The variables may be local to a single machine or shared by multiple machines. It can be classified in the models known as "extended finite state machines."

The global state analysis of protocols usually generates a very large number of states. A previous work [BULB93] on reachability analysis, automated the analysis of communication protocols. This analysis was based on the exhaustive search method. The main restriction with this method is its inability to continue processing in the face of the "state space explosion."As stated in [HOLZ91], an estimate for the maximum size of the state space that can be reached for a full reachability analysis is about 10^5 states. A protocol with more than 10^5 states cannot be fully analyzed utilizing the exhaustive search method, due to computer memory limitations. A controlled partial search method "*Supertrace*" was thus introduced in [HOLZ91] to analyze protocols which cannot be analyzed by the exhaustive search method. The *Supertrace* is implemented in this thesis.

A conformance test is used to ensure that the external behavior of a protocol's implementation is equivalent to its formal specification. In conducting a conformance test, we are given a known protocol specification and an unknown implementation. The implementation, for practical purposes, is considered a "black box" with a finite set of inputs and outputs. The test provides a sequence of input signals, and observes the resulting outputs. The *implementation under test* (IUT) should pass the test only if all observed outputs match those prescribed by the formal specification. The series of input sequences which are used to exercise the protocol implementation in this way are referred as *conformance test sequence* throughout this thesis.

A previous study [MILL90] on this issue observed gaps between the specification, the verification, and the conformance testing of network protocols. Protocol models which are designed for specification purposes usually have many powerful program language constructs, to simplify the specification, but are difficult to analyze. Protocol models designed primarily for analysis

purposes, such as the CFSM model, are too simple for the specification of modern, complex protocols. Recent works on conformance testing have started from the description of a protocol as an incompletely specified finite state machine with input/output labels on the transitions [CHEN90],[DAHB90]. Protocol specifications are not normally described in this manner.

Suppose a test designer was required to test a protocol specified using a formal language (i.e. Estelle). First, the specification must be translated to an I/O diagram. This is a labor intensive complex process, and during which errors are easily introduced. Only, when this translation is complete, can the designer begin to generate the inputs for conformance testing.

A procedure, created in [LUND90A], is implemented in this thesis, for the generation of a test sequence for a protocol specified in the SCM model. The purpose was to reduce the work and the possibility of error, for the designer. The automation of the conformance test sequence generation is also an attempt to close the gap between specification/verification and testing of protocols. In this thesis, the test generation starts from a protocol model, designed for the specification and verification of protocols. The procedure [LUND90A] and its automation as a software tool does not guarantee that all the errors or combination of errors in a protocol are found. But they do represent an attempt to exercise all parts of the protocol, providing some assurance that the implementation meets its purpose.

B. Scope Of Thesis

The scope of this thesis is two fold: The first is to present implementation of the Supertrace algorithm, apllied to the CFSM and SCM protocol models. This leads to the reachability analysis of larger protocols formally specified by CFSM and SCM models that cannot be totally analyzed by using exhaustive search methods. An earlier study on this issue is capable of generating reachability analysis of protocols that are small enough to be analyzed by full state space search method. This thesis expands this work to cover the analysis of bigger protocols by a controlled partial search method known as "*Supertrace*" algorithm. The output of the program was compared to several previous works and was consistent with their results.

The second part of this thesis is on testing protocol implementations. A software tool that automates the generation of a testing sequence is introduced for testing and verification of network protocols. The procedure implemented in this program was created in [LUND90A].

When combined with the earlier work a protocol can be specified as a system of communicating machines, analyzed by the mushroom program and a set of "conformance tests" can be generated from to insure that an implementation of the protocol is, to some degree at least, in conformance with its specification

C. Organization

This thesis has six chapters. Chapter II reviews the Communicating Finite State Machines (CFSM) and System of Communication (SCM) models. Chapter III describes the Super Trace algorithm and introduces two programs based on the algorithm. The **Simple Mushroom With Supertrace** and **Big Mushroom With Supertrace**, expand the automation of the global reachability analysis of larger protocols formally specified by CFSM and SCM models respectively.

In Chapter IV, a procedure for generating test sequences for a formally specified protocol is introduced and a software tool that automates this process is described.

In Chapter V, examples of the use of software tools are given.

Chapter VI concludes the thesis with a research review and suggestions for future work.

II. INTRODUCTION TO CFSM AND SCM MODELS

A. Communicating Finite State Machines

Communicating finite state machine (CFSM) model is a simple model which requires that each machine in the network is modeled as a finite automaton or finite state machine (FSM). The Communication channels between pairs of machines are modeled as one-way, infinite length FIFO queues. There is a great deal of literature on this model [PENG91][RUDI86][VUON83]. The model is defined for an arbitrary number of machines. A two machine model (shown in Figure 1) will be presented in this chapter for simplicity.

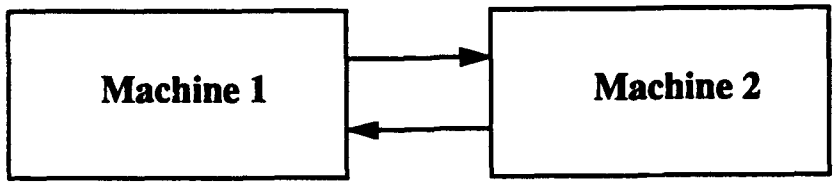

Figure 1 : CFSM, Two machine model representation

1. Model Definition

This section defines the CFSM model [GOUD83] and provides a simple protocol specification and analysis to clarify the definition.

A *communicating machine M* is a finite, directed labeled graph with two types of edges, *sending* and *receiving*. A sending (receiving) edge is labeled '*-g*' ('*+g*') for some *message g*, taken from a finite set *G* of messages. One of the nodes in *M* is identified as the *initial node* by some directed path. A node in *M* whose outgoing edges are all sending (receiving) edges is a *sending (receiving)* node; otherwise the node is a *mixed* node. The nodes of M are often referred to as *states*; these two terms will be used interchangeably throughout this thesis.

Let *M* and *N* be two communicating machines having the same set *G* of messages the pair *(M,N)* is a network. A global state of this network is a four tuple $[m, c_m, n, c_n]$, where m and n are nodes (states) from *M* and *N*, and c_m and c_n are strings from the set *G* of messages. Intuitively, the global state $[m, c_m, n, c_n]$ means that the machines *M* and *N* have reached states *m* and *n*, and the communication channels contain the strings c_m and c_n of messages, where c_m denotes the messages sent from M to N in channel C_M, and c_n denotes the messages sent from *N* to *M* in channel C_N. In the case of say k number of machines where $k > 2$ the global state can be represented as

$[m_1,q_{12},q_{13},\ldots,m_2,q_{21},q_{23},\ldots,m_3,q_{31},q_{32},\ldots,\ldots,m_k,q_{k1},q_{k2},\ldots]$ where m_i's are the nodes of machines M_i and q_{ij} contains the messages sent from M_i to M_j. Subscripts i and j ranges from $1..k$ and $i \neq j$.

The initial global state of (M,N) is $[m_0,E, n_0,E]$, where m_0 and n_0 are the initial states of M and N, and E is the empty string.

The network progresses as transitions are taken in either M or N. Each transition consists of a state change in one of the machines, and either the addition of a message to the end of one channel (sending transition) or the deletion of a message from the front of one channel (receiving transition).

A sending transition in M (N) adds a message to the end of channel C_M (C_N); a receiving transition in M (N) removes a message from the front of channel C_N (C_M).

Suppose $+g$ is a receiving transition from state i to j in machine M (N). The transition can be executed if and only if M (N) is in state i and the message g is at the front of the channel C_N (C_M). The execution takes zero time. After its execution, machine M (N) is in state j, and the message g has been removed from the channel C_N (C_M).

Similarly, suppose $- g$ is a sending transition from state i to j in machine M (N). The transition can be executed if and only if M (N) is in state i. Afterwards, g appears on the end of the outgoing channel, and the machine has transitioned to state j.

Suppose $s_1= [m, c_i, n, c_j]$ is a global state of (M,N). State s_2 *follows* s_1 if there is a transition (in M or N) which can be executed in s_1 if there is a sequence of states $s_i, s_{i+1},\ldots, s_{i+p}$ such that s_i follows s_1, s_{i+1} follows s_i, and so on, and s_2 follows s_{i+p}. A state s is *reachable* if it is reachable from the initial state.

The communication of a network (M,N) is a directed graph in which the nodes correspond to the reachable global states of (M,N), and the edges represent the *follows* function. That is, there is an edge from state s_i to state s_j if ad only if s_j follows s_i. The edges are labeled with the transitions which they represent. This reachability graph can be generated by starting with the initial state, and adding the states which follow it, connecting them to it with edges; and repeating for each new state generated.

The next two definitions are of errors that may occur in a communication protocol which are detectable by analysis.

A global state $[m, c_m,n, c_n]$ is a *deadlock state* if both m and n are receiving nodes and $c_m=c_n=E$, where E denotes the empty string.

A global state $[m, c_m,n, c_n]$ is an *unspecified reception state* if one of the following two conditions is true:

(1) m is a receiving state, the message at the head of channel c_n is g, and none of m's outgoing transitions is labeled '+g.'

(2) n is a receiving state, the message at the head of the channel c_m is g, and none of n's outgoing transitions is labeled '+g.'

These error conditions can be identified by generating the reachability for a network, and inspecting all states as they are generated. In the next section, an example protocol is specified and analyzed using CFSM model.

2. An Example Of Protocol Specification And Analysis Using CFSM Model

A simplified version of the Stop-and-Wait data link protocol will be analyzed as an example of analysis with CFSM model. The interface between user and data link layer are assumed to be error free and higher layer passes information/frame without error to the Data link layer. At data link layer this protocol consist of two machines a sender and a receiver. In Figure 2, machine 1 serves as the sender and machine 2 serves as the receiver.

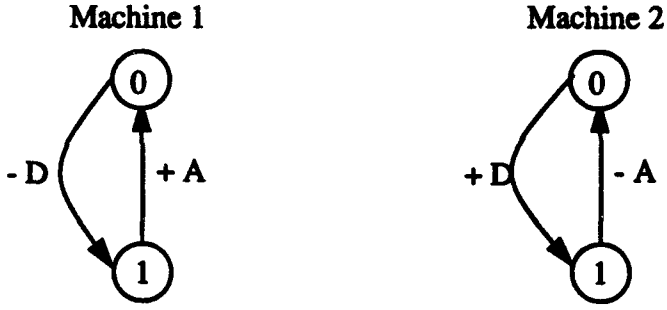

Figure 2 : CFSM Specification for Stop-and-Wait

The sender places a frame on the channel for the receiver. The receiver senses a frame on the incoming channel and accepts and removes the message from the channel. The receiver then sends an acknowledgment packet to the sender. The sender receives the acknowledgment packet and is able to send another frame of information to the receiver.

The -D and +D represents the sending and receiving of data respectively. The -A, and +A represent the sending and receiving acknowledgment respectively. Since the initial state of each machine is 0; the initial global state is [0,E,0,E].

The reachability analysis can be done by a simple procedure. Starting with the initial global state only one transition is possible, the -D of machine 1 from state 0. This leads to global state [1,D,0,E]. We can continue the analysis in the same manner detecting the possible transitions from this global state until possible global states are found. The complete reachability analysis

consisting of four states is given in Figure 3. There are no deadlocks or unspecified receptions in this protocol.

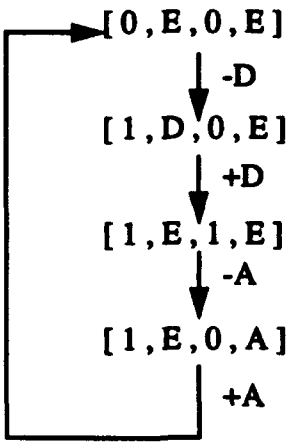

Figure 3 : Reachability Analysis of Stop-and-Wait protocol

Another CFSM specification of an imaginary network protocol consisting of three communicating machines is shown in Figure 4.

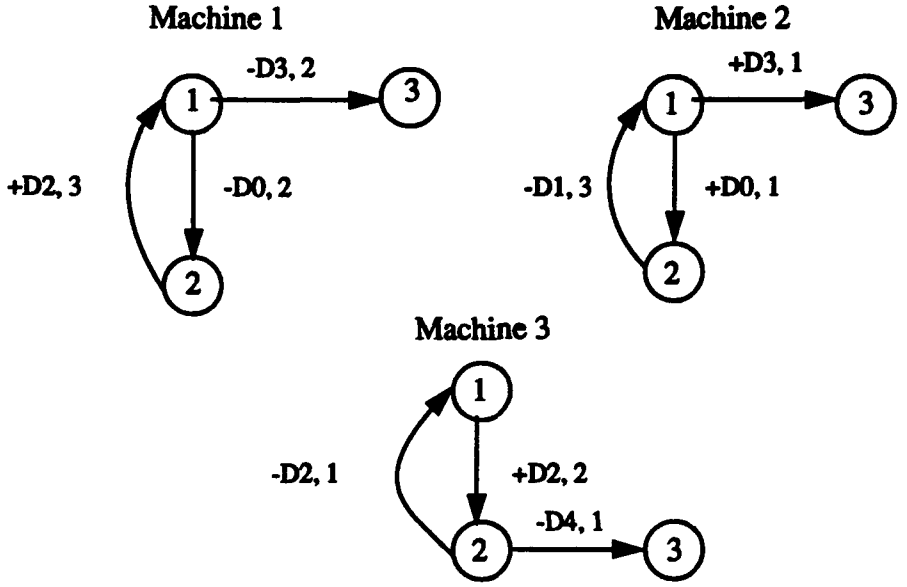

Figure 4 : CFSM Specification of Example protocol

The directed edges are labeled such that the character-number combinations following the '-/+' shows the messages and the numbers at the end represent the destination machine. A clockwise ring is formed with each machine sending one message to the next machine and receiving a message from the previous machine. The initial state of each machine is 1; thus the initial global

state is [1,E,E,1,E,E,1,E,E]. The reachability analysis of this protocol shown in Figure 5. In this analysis there is one deadlock condition and one unspecified reception. In global state [3,E,E,3,E,E,1,E,E], all the channels are empty and all the nodes are receiving nodes satisfying the deadlock condition. In global state [2,E,E,1,E,E,3,D4,E], machine 1 and machine 2 are in receiving states but none of the outgoing transitions are labeled '+D4', satisfying an unspecified reception condition.

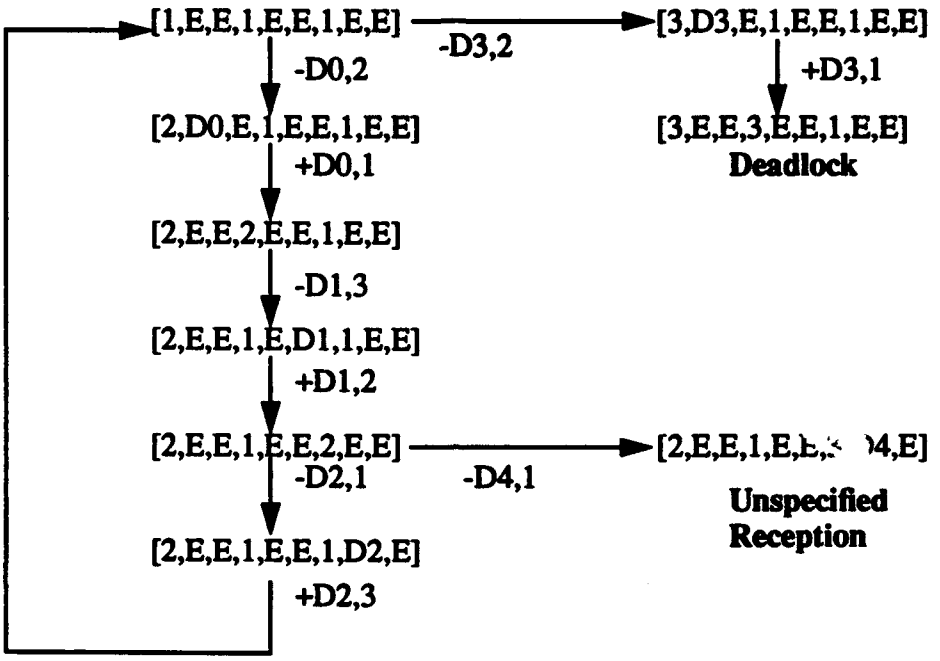

Figure 5 : Reachability Analysis of Example protocol

3. Summary

The CFSM model is simple and easy to understand. However, as the protocols become more complex, this model becomes difficult to use due to a combinatorial explosion of states. The analysis might not terminate if the queue length is unbounded. The number of states in the reachability graph will be unmanageably large for such complex protocols even if the queue length is bounded. A computer analysis might eventually terminate, but still the CPU time would be days even months, obviously impractical.

Another disadvantage is that as the protocols become more complex, the specification of the protocol can be so large, consisting of many states and transitions, that makes it very hard to understand if it is the intended specification. Several examples are given in Chapter V that shows the largeness of analysis output for some protocols.

B. Systems Of Communicating Machines

In this section the SCM model is described. First the model definition is given, then the algorithm for generating the system state analysis is described. Finally, to illustrate the important aspects of the model it is used to specify analyze a sample protocol.

1. Model Definition

A *system of communicating machines* is an ordered pair $C= (M,V)$, where

$M= \{m_1,m_2,...,m_n\}$

is a finite set of *machines*, and

$V= \{v_1,v_2,...,v_k\}$

is a finite set of shared variables with two designated subsets R_i and W_i specified for each machine m_i. The subset R_i of V is called the set of read access variables for machine m_i, and the subset W_i the set of write access variables for m_i.

Each machine $m_i \in M$ is defined by a tuple (S_i,s,L_i,N_i,τ_i), where

(1) Si is a finite set of states;

(2) $s \in S_i$ is a designated state called the *initial state* of m_i;

(3) L_i is a finite set of *local variables*;

(4) N_i is a finite set of names, each of which is associated with a unique pair (p,a), where p is a predicate on the variables $L_i \cup R_i$, and a is an action on the variables of $L_i \cup R_i \cup W_i$. Specifically, an action is a partial function

$$a: \quad L_i \times R_i \rightarrow L_i \times W_i$$

from the values of the local variables and read access variables to the values of the local variables and write access variables.

(5) τ_i: $S_i \times N_i \rightarrow S_i$ is a transition function, which is a partial function from the states and names of m_i to the states of m_i.

Machines model the entities, which in a protocol system are processes and channels. The shared variables are the means of communication between the machines. Intuitively, R_i and W_i are the subsets of V to which m_i has read and write access, respectively. A machine is allowed to make a transition from one state to another when the predicate associated with the name for that transition is true. Upon taking the transition, the action associated with that name is executed. The action changes the values of local and/or shared variables, thus allowing other predicates become true.

The sets of local and shared variables specify a name and range for each. In most cases, the range will be a finite or countable set of values. For proper operation, the initial values of some or all of the variables should be specified.

A system state tuple is a tuple of all machine states. That is, if (M,V) is a system of n communicating machines, and s_i, for $1 \leq i \leq n$, is the state of the machine m_i, then the n-tuple $(s_1,s_2,...,s_n)$ is the system state tuple of (M,V). A *system state* is a system state tuple, plus the outgoing transitions which are enabled. Thus two system states are equal if every machine is in the same state, and the same outgoing transitions are enabled.

The global state of a system consists of the system state tuple, plus the values of all variables, both local and shared. It may be written as a larger tuple, containing the system state tuple with the values of the variables. The initial global state is the initial system state tuple, with the additional requirement that all variables have their initial values. The initial system state is the system state such that every machine is in its state, and the outgoing transitions are the same as in the initial global state.

A global state corresponds to a system state if every machine is in the same state, and the same outgoing transitions are enabled. Clearly, more than one global state may correspond to the same state.

Let $\tau(s_1,n) = s_2$ be a transition which is defined on machine m_i. Transition τ is enabled if the enabling predicate p, associated with name n, is true. Transition τ may be enabled whenever mi is in state si and the predicate p is true (enabled). The execution of τ is an atomic action, in which both the state change and the action a is associated with n occur simultaneously.

It is assumed that if a transition is enabled indefinitely, then it will eventually occur. This is an assumption of fairness, and is needed for the proofs of certain properties.

2. Algorithm: System State Analysis

The process of generating the set of all system states reachable from the initial state is called system state analysis. This analysis construct a graph, whose nodes are the reachable system states, and whose arcs indicate the transitions leading from each system state to another. This graph may be generated by a mechanical procedure which consist of the following three steps [LUND91];

1. Set each machine to its initial state, and all variables to their original values.The initial set of reachable system states consists of only the initial system state; the initial graph is a single node representing this case.

2. From the current system state vector and variable values, determine which transitions are enabled. For each of these transitions determine the system state which results from its execution. If this state (with the same enabled transitions) has already been generated, then draw an arc from the current state to it, labeling the arc with the transition name. Otherwise, add the new

system state to the graph, draw an arc from the current state to it, and label the arc with the name of the transition.

3. For each new state generated in step 2, repeat step 2. Continue until step 2 has initial, been repeated for each system state thus generated, and no more new states are generated

3. An Example Protocol Specification and Analysis Using SCM Model

The stop-and-wait protocol is also used to demonstrate the analysis using SCM model. The specification of the stop-and-wait protocol as represented by SCM model is shown in . The specification consists of two finite state machines, the local and shared variables, and the predicate action table, Table 1. The local variables are in_buff and out_buff shown under their corresponding FSMs. The shared variables are: CHAN and RET and shown between the two machines. The initial state of each machine is 0, with the shared and local variables are empty except the local variable out_buff which has "D." The 'D' in out_buff represents data and characters 'E' and 'A' in predicate action table represent empty string and acknowledgment respectively.

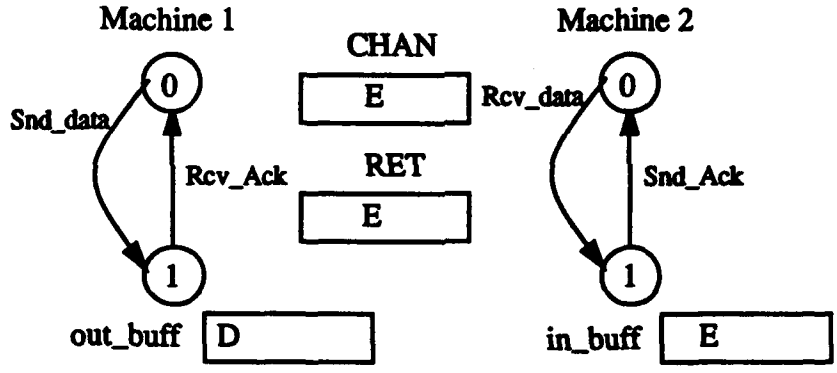

Figure 6 : SCM Specification of Stop-and-Wait Protocol with Variables

TABLE 1: PREDICATE ACTION TABLE FOR STOP-AND-WAIT PROTOCOL

Transition	Enabling Predicate	Action
Snd_data	CHAN = E ∧ out_buff ≠ E	CHAN:= out_buff out_buff:= E
Rcv_Ack	RET = A	RET:= E ; CHAN:= E
Rcv_data	CHAN ≠ E	in_buff:= CHAN
Snd_Ack	TRUE	RET:= A ; in_buff:= E

For this example the assumption is made that data is always made available to the CHAN from out_buff. The global reachability analysis, shown in Figure 7, has 4 states. The format for the global state tuple is:

[Machine1_state, out_buff, Machine2_state, in_buff, CHAN, RET]

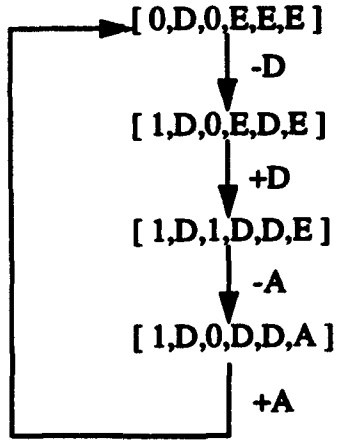

Figure 7 : Global Reachability Analysis of Stop-and-Wait Protocol

The system state analysis for the stop-and-wait protocol also has 4 states (see Figure 8). For more complex protocols, there may be a big difference between global and system states. For example a sliding window protocol with a window size of 8 the system state analysis was shown to generate 165 states, while the full global analysis generated 11880 states [LUND91].

The format for a system state tuple analysis is:

[Machine1_state , Machine2_state]

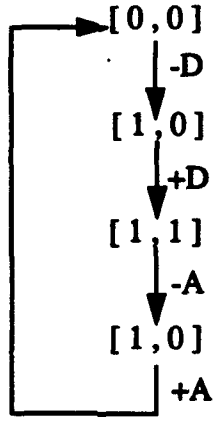

Figure 8 : System Reachability Analysis of Stop-and-Wait Protocol

4. Summary

The SCM model has desirable properties which overcome some of the disadvantages of the CFSM model. One of the advantages of the SCM model is that it significantly reduces the state explosion through the use of system state analysis. In some cases, however the system state analysis is not sufficient for protocol analysis. Some other method - such as global analysis must be performed. A problem is that loops in the state machines may cause an insufficient system state analysis.

Another advantage of SCM model is that it allows communication between machines in nonsequential manner, unlike a FIFO queue representation in the CFSM model. The SCM model specification is easier to understand than the CFSM model for more complex protocols

III. SUPERTRACE ALGORITHM

A. The Idea Behind The Supertrace Algorithm

The standard full, or exhaustive, search algorithm explores all reachable composite system states for a set of interacting finite state machines. Every reachable state and every sequence of reachable states can be checked for a set of correctness criteria such as deadlock condition and unspecified reception. However, the size of the search space and the limits of physical memory severely restrict the use of this method. If the size of the state space is R and the maximum number of states that can be stored in memory during the search is M both the coverage and the search quality can only reach 100% when $R \leq M$. When $R > M$ the coverage reduces to M/R, but the search quality is likely to be worse.

To give an idea of the magnitude of such a search consider the following example. Suppose that we have a protocol for two machines, each with 100 states, one message queue, and five local variables. The two message queues are restricted to five slots each, and the range of values for local variables are assumed to be limited to ten values. The number of distinct messages exchanged is 10. In this sample system, there are $10^{5 \cdot 2}$ possible states of the protocol variables. Each process can be in one of 10^2 different states, so two processes can maximally be in 10^4 different composite system states. Finally each queue can hold up to five messages, where each message can be one out of ten permutations. The total number of system states in the worst case is

$$10^{10} \cdot 10^4 \cdot \left[\sum_{i=0}^{5} 10^i \right]^2$$

or in the order of 10^{24} different states. If each state could be encoded in 1 byte of memory and analyzed in 10^{-6} sec, it would still require at least 10^{15}times more memory as currently available on most systems, and would take roughly 10^{11} years to perform an exhaustive analysis.

Fortunately, the number of effectively reachable states is usually much smaller than the total number of states calculated above. Even relatively small protocol systems, however, can easily generate up to 10^9 reachable states. Therefore the full search method is feasible only if we can reduce the complexity of our models to the maximum that a given machine can analyze.

If the state space is larger than the available memory can accommodate, the exhaustive search strategy discussed above reduces to a partial search, without guaranteeing that the most important parts of the protocol are inspected. This observation has led to the development of a new class of algorithms that exploits the benefits of partial search.

One of the most effective partial search methods is the "Supertrace Algorithm" [HOLZ91], which is implemented in this thesis.

1. Supertrace Algorithm (A Controlled Partial Search Method)

In this section the idea behind the supertrace will be discussed as it is introduced in [HOLZ91].

Let *A* represent our state space set and *M* the bytes of memory available.The standard way to maintain the state space set *A* is using a technique called hashing. Redundant states are restricted from set *A* by means of a hashing function.

Each state is placed into a hashing table based on their hashing value $h(s)=i$ where *h* is the hashing function, *s* is the global state, and *i* is the index for the hash lookup table (see Figure 9).

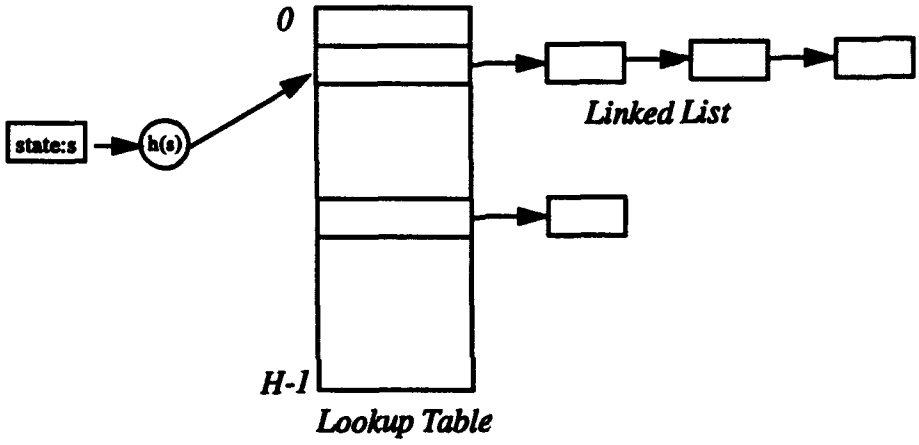

Figure 9 : Hash Lookup Table

If we have *H* slots in the hash lookup table. Hash function $h(s)$ must be defined such that it returns arbitrary value *i* in the range $0..(H-1)$. But the possibility exists that two different states produce the same hash value. In the case of a large protocol the hash table will have to accommodate a large number of states. When $A > H$ the hash function will always produce some duplicates indices values of *i* for an average of A/H different states. To accommodate these duplicate index values we use an open hash and all states that hash to the same value are stored in a linked list that is accessible via the lookup table under the calculated index. When the table is full, each new state must be compared to average A/H other states before it can be inserted into the linked list or discarded as redundant. As *A* continues to grow beyond the first *H* states, the number of comparisons required increases steadily, and the search efficiency degrades. There is a time penalty for analyzing systems of more than *H* states. This type of hashing was used for analysis of protocols in previous work [BULB93].

We want to make H as big as possible or at least 10^3 times bigger than we expect A to be. If we can have $H \gg A$ then there will be very few, if any, conflicts. In this case we do not need to store complete state descriptions in the hash table: in all but a few cases the hash value $h(s)$ uniquely identifies a state. A single bit of storage will suffice to verify if a state has already been generated.

If we have M bytes of memory available, assuming 8 bits per byte we have $8M$ bits for state space. The state is not stored. Since no state is stored, memory efficiency is greatly increased and there are no states to compare a new state against. The bit position in the hash table uniquely identifies the state. The method can be expected to work well if the state space is sparse and indeed H is very large. For $H \gg A$ hash conflicts are rare. When $A > H$ then conflicts will occur. The accuracy of our analysis will depend upon the percentage of hash conflicts. Because of hash conflicts some deadlocks or unspecified receptions may go undetected.The method therefore approximates an exhaustive search for smaller protocols and slowly changes into a controlled partial search method for larger protocols.The Supertrace Algorithm as compared to the exhaustive search can not guarantee 100% coverage due to possibility of unresolved hash conflicts. The implementation of the "Supertrace Algorithm" will be explained in the following sections.

B. Simple Mushroom With Supertrace

The first program to be examined is called **Mushroom with Supertrace.** It was written in the Ada programming language. **Mushroom** was written to automate the reachability analysis of protocols specified by the CFSM and SCM models [BULB93]. The Mushroom with Supertrace was developed to extend the applicability of Mushroom program to larger and more complex programs. There are actually two separate versions. The first called, simple mushroom with supertrace, analyzes the CFSM models. The second version analyzes the SCM models, either as system state analysis (smart mushroom), or a full global analysis (big mushroom with supertrace) of a protocol specified formally by the SCM model. The Supertrace algorithm is not implemented for smart mushroom program since the state space generally does not grow beyond the limits of memory. The General structure of mushroom program is shown in Figure 10.

The explanation, Simple Mushroom with Supertrace, is divided into four sections: program structure, inputs, reachability analysis, and outputs. The portions of this program that are common to the original Mushroom program along with the details of the mushroom program are not discussed.

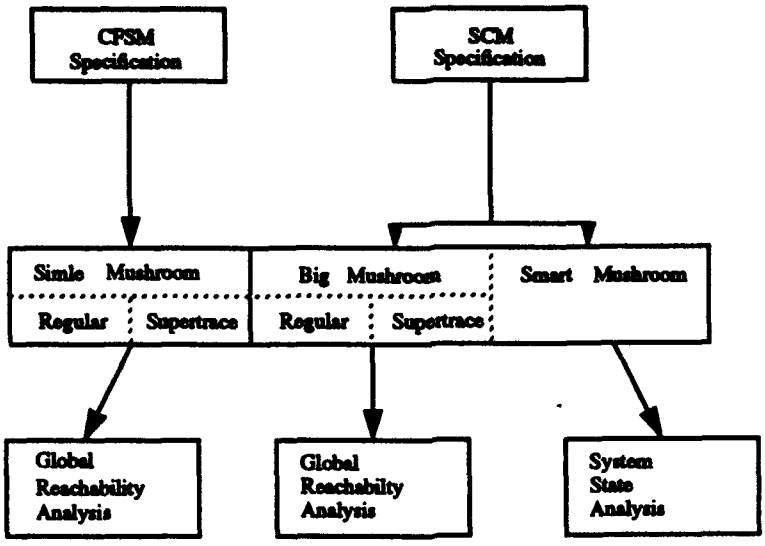

Figure 10 : General Structure of Mushroom Program

1. Program Structure

The Simple Mushroom program consists of Ada subprograms (procedures and functions), which are separate compilation units and subunits of compilation units. Related subprograms are also gathered in the same files. The compilation units of the program are shown in Table 2. Procedure *main* is the parent unit. All of the subprograms are the subunits of procedure *main* [ANSIMIL93].

TABLE 2: SIMPLE MUSHROOM COMPILATION UNITS

Compilation Unit	Description	File Name
main(procedure)	This is the parent unit. Contains the main data structures, global variable and the driver.	tmain.a
load_machine_array (procedure)	Builds the adjacency lists from FSMs.	tinput.a
read_in_file(procedure)	Parses the input FSM text file	tinput.a
build_Gstate_graph (procedure)	Generates the reachability graph.	treachability.a
IsEqual (function)	Compares two global states for equality	treachabilty.a
hash(function)	Generates an index number according to the hashing function	treachabilty.a
clear_pointers(procedure)	Deallocates the dynamic memory space for another analysis	treachability.a
Print Queue(procedure)	Prints the FIFO queues	toutput.a

TABLE 2: SIMPLE MUSHROOM COMPILATION UNITS

Compilation Unit	Description	File Name
output_Gstate_transition (procedure)	Outputs the transition name	toutput.a
output_Gstate_node (procedure)	Outputs the machine states, unspecified receptions, and the states with deadlocks.	toutput.a
output_machine_arrays (procedure)	Outputs the FSM description in a tabular format	toutput.a
output_unexecuted_transitions (procedure)	Outputs the unexecuted transitions	toutput.a
create_output_file (procedure)	Creates an output file for storing the analysis results	toutput.a
output_analysis(procedure)	Driver for the output subprograms	toutput.a
system_call(procedure)	Interface procedure for Unix system calls via C.	tsystem.a
message_queues (package)	Implements the queue operations for the FIFO communication channels.	tqueues.a
pointer_queues (generic_package)	Implements the queue operations for the pointer queue that stores the global tuples temporarily	tqueues_2.a

2. Input

The CFSM specification of a protocol consists of only FSMs of the communicating machines. FSMs are represented with a text file. The user enters the directed graphs as a text file using some reserved words, numbers, and characters. For the list of reserved words the reader should refer to [BULB93]. The maximum number of machines allowed is eight, and the number of states for each machine can be from 0 to 50. Transition names must be at most three characters long and may be any combination of letters or digits. These constraints can be relaxed with modifications to the program, if necessary.

The input file for the *stop-and-wait* protocol in Chapter II for the CFSM model is shown in Figure 11. The reserved word "state" represents the states of the machine that they come after. For example "trans -D 1 2" (first line at state 1 in machine 1) represents a transition from state 0 to state1 by sending D to machine 2. The first character '-' or '+' following reserved word "state" represents sending or receiving data respectively. "Initial_state 0 0" means that the initial states of machine 1 and machine 2 are state 0.

First, this file is parsed by read_in_file procedure and tokens are generated. Then, Load_machine_array procedure constructs an adjacency list which represents the FSMs.

```
start
number_of_machines 2
machine 1
state 0
trans -D 1 2
state 1
trans +A 0 2
machine 2
state 0
trans +D 1 1
state 1
trans -A 0 1
initial_state 0 0
finish
```

Figure 11 : Text File Description of Stop-and-Wait protocol

The adjacency list for the stop-and-wait protocol is depicted in its structural form in Figure 12. This adjacency list is used for constructing the global reachability graph. The adjacency list contains all the necessary information for generating the global reachability graph.

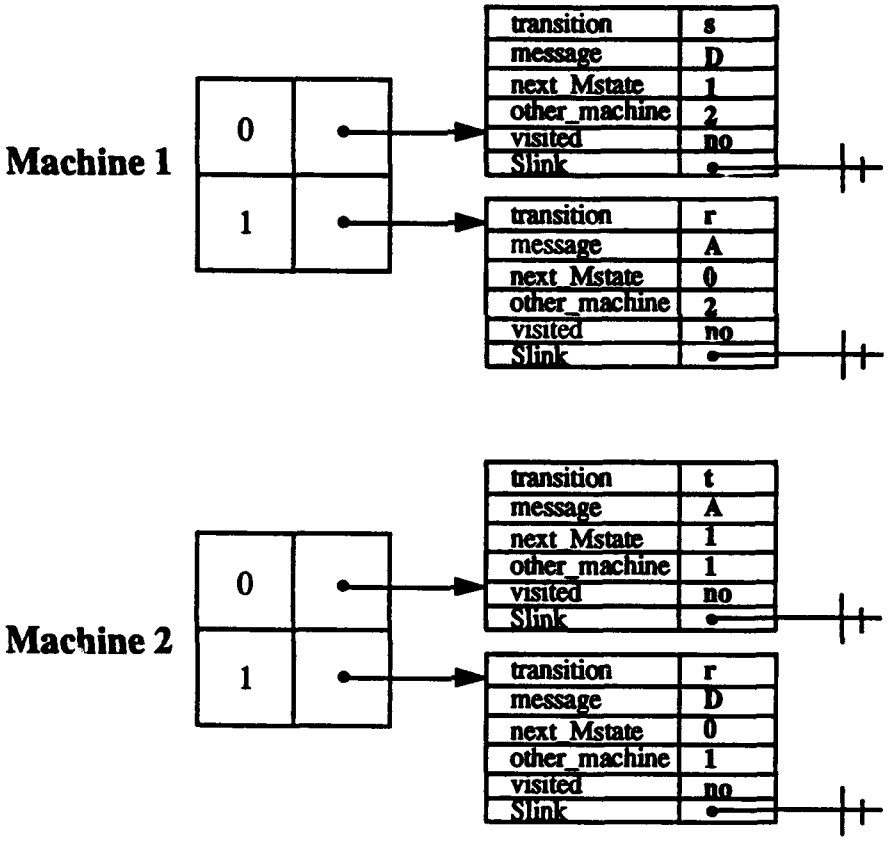

Figure 12 : Adjacency list for the example Stop-and-Wait protocol

3. Reachability Analysis

After reading the input file the program generates the global reachability graph. It uses the adjacency list and the initial state to begin construction the global reachability graph. Starting with the initial state new states are generated and compared with previous ones based on their respective index value. The global reachability graph construction algorithm is given in Figure 13.

```
loop (main loop)
  for index1 in 1 .. total_number_of_machines loop
    place_holder(index1) := machine_array(index1)(Mstate(index1))
    while (place_holder(index) /= null) loop
      loop
        if (place_holder(index1).transition = s) then
          Enqueue the message into the corresponding message queue
          search hash look-up table for this global state tuple
          if slot of the hash look-up table was not set then
            This is assumed to be a new state set the slot and create a new state
            Enqueue this new node to the pointer_queue
          else
            print out the transition and discard the tuple
          end if
        else
          if (place_holder(index1).transition) = r and at least one of the message queues for
            this machine is not empty then
            find this message queue and Dequeue
            search hash look-up table for this new global state tuple
            If slot of the hash look-up table was not set then
              This is assumed to be a new state set the slot and create a new node
              Enqueue this new node to the pointer_queue
            else
              print out the transition and discard the tuple
            end if
          end if
        end if
        place_holder(index1) := place_holder(index1).Slink
        exit;
      end loop
    end loop
    if pointer_queue empty then
      exit
    else
      Dequeue pointer_queue and update M_state for this new node
    end if
end loop (main loop)
```

Figure 13 : Algorithm for Generating Global Reachability Graph for CFSM

During the graph construction, the program also detects the global states with dead locks or unspecified receptions. The program also finds the maximum message queue size and channel overflows. Analysis results are stored in an output file. This avoids the need to transverse the entire graph an additional time at the end of the program. Program run time is thus dramatically reduced.

One of the most time consuming procedures is the search algorithm used to detect if a state was previously created. The previous version of this program used open hashing to search through the previously created global states. All states were kept in a linked list associated with their hash index. For the analysis of small protocols this is not a problem. The search is fast, the memory required is small, and the linked lists are short. The analysis of larger protocols, link lists grows longer due to increased hash conflicts and the applicability of regular mushroom becomes restricted.

With Supertrace the search is also made via hash function but utilizes a different implementation. First, the size of hash table is determined based upon the expected number of the states generated, to ensure adequate coverage, but is limited by the availability of memory. Second, the hash function uses the machine states and the messages on the queues between the machines to provide a fast and efficient mapping. The complexity of the search algorithm is always O(1). This is obvious when the hash function generates a unique index (no collision). When the hash function generates the same index for two different states Supertrace, discards the new state, (as a duplicate) as it only checks if the hash table slot is set(collusion) or not set(new state). Previous tuples are not compared. This makes the search more efficient. Because we are using a very big hash table, the hash function creates a distinct index (table slot) for almost every global state.

The effectiveness of the Super Trace algorithm depends upon the ratio of hash table size to the expected number of states, the effectiveness of the hash function which generates the indices for the hash array. The hash function which generates the indices for protocols specified in CFSM model is shown in Figure 14.

The second issue that has effect on Supertrace Algorithm's efficiency is the available memory on the system. The size of the hash table must be as big as possible to minimize the number of hash conflicts. The need for a very large memory can not be overemphasized.

The impact of such a large table is minimized by utilizing the Ada Programming Language predefined pragma "pack." The pragma "pack" tells that storage minimization should be main criterion for representing of the given type (hash-lookup-table) to the compiler. By using that option, boolean types which normally are represented as 1 byte (8 bits) in the memory; can be reduced to one bit which saves seven bits per byte. We can effectively increase the size of our hash table by 700% without using additional memory space. So a hash table of size 1545278 is used in our applications without using big part of memory.

The structure of a global state is shown in Figure 15. The maximum number of outgoing transitions is artificially limited to 7. It can be increased if necessary. A maximum channel capacity of 6 messages is introduced to ensure that the analysis eventually stops.

```
function hash (m : in machine_state_array;
               q : queue_type) return integer is

  index : integer := 0;
  sum   : integer := 0;
begin
  for i in 1..8 loop
    for j in 1..8 loop
      if integer(q(next_machine_type(i),next_machine_type(j)).tail) /=0 then
        for l in 1..integer(q(next_machine_type(i),next_machine_type(j)).tail) loop
          for k in 1..3 loop
            sum :=sum+character'pos(q(next_machine_type(i),next_machine_type(j)).store(l)(k))*j;
          end loop;
        end loop;
      end if;
    end loop;
  end loop;
  index := (integer(m(8))*19765)+(integer (m(7))*2978) + (integer(m(6))* 43270)
           +(integer (m(5))*13791) + (integer(m(4))* 28433) +(integer(m(3))* 17237)
           +(integer (m(2))* 37777) + (integer(m(1))* 635799);
  return ((index+sum*4)mod 30545423);
end hash;
```

Figure 14 : Example Hash Function For Stop-and-wait Protocol

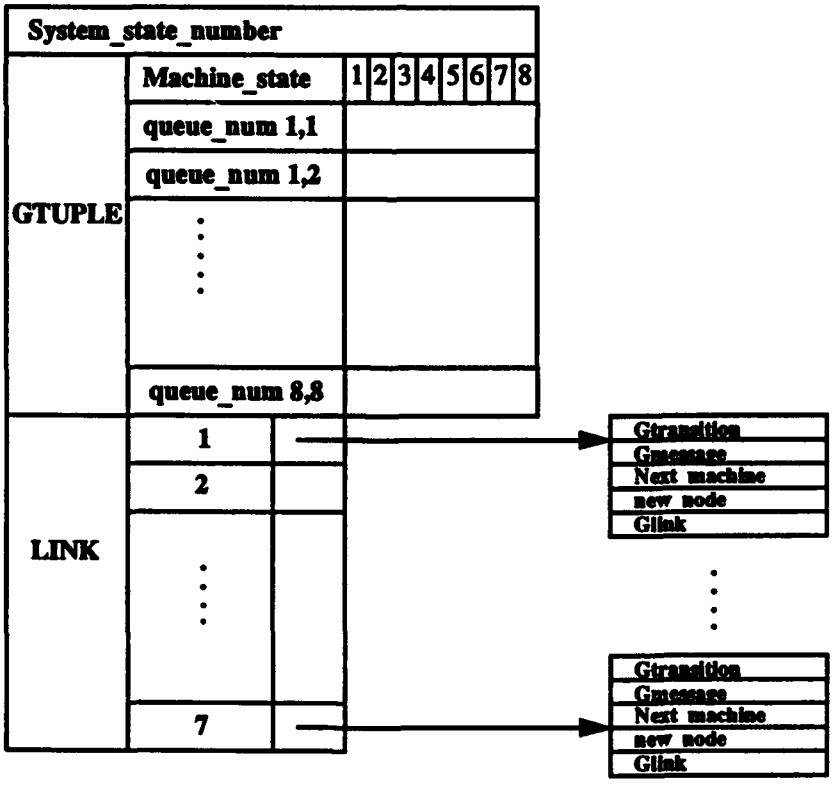

Figure 15 : Global State Structure with outgoing transitions

4. Output

The program stores the analysis results in a file named by the user during the reachability graph construction. The file contains the specification in a tabular format, the reachability graph and the results of the analysis. The analysis results consists of six separate sections. They are the number of states generated, number of states analyzed, number of deadlocks detected, number of unspecified receptions detected, maximum message queue size and the number of channel overflows. Global states with deadlocks and unspecified receptions are also marked in the reachability graph. The output file also lists any unexecuted transitions.

The program output for the imaginary protocol in Chapter II is listed in Figure 16. Since no states are stored, in case of a collision we can not determine whether it is a hash conflict of a new state or a duplicate state. These states are referred as 0 in the output file. For example, In our example protocol after *state 8* "+d2" transition is taken which leads to state 1. Since program doesn't keep state 1 it will just output 0 for the duplicate state.

C. Big Mushroom With Supertrace

In this section, the program that automates the full global analysis (*big mushroom*) for a protocol specified by a SCM is model described. The description of the program is divided into four sections: general program structure, inputs to the program, generating the reachability graph, and outputs of the program. Since the *smart mushroom* program mentioned in Chapter II generates a relatively small number of states it is considered outside the scope of this thesis and will not be mentioned in the following sections.

1. Program Structure

Program structure of *Big mushroom* is similar to the structure of *Simple Mushroom*. The SCM model specification is more complicated than the CFSM specification, but this complexity in the specification brings some advantages to the analysis as mentioned in Chapter II. A protocol specified by the SCM model consists of FSMs, variable definitions, and predicate-action table, rather than just the FSMs as in CFSM model.

FSMs are entered into the program in the same manner as in *the Simple Mushroom* program using a text file. The variable definitions and predicate-action table must also be entered into the program. The user enters these parts by completing Ada packages and subprograms using the templates provided.

The compilation units for the program are shown in Table 3. The user has access to the last four packages/subprograms. Once the user completes these programs using the templates and

```
                    REACHABILITY ANALYSIS of : example.fsm
                                SPECIFICATION

                    ---------------------------------------------
                    |       Machine  1  State Transitions       |
                    ---------------------------------------------
                    | From  |  To  | other machine | Transition  |
                    ---------------------------------------------
                    |   1   |  2   |       2       |   s  d0     |
                    |   1   |  3   |       2       |   s  d3     |
                    |   2   |  1   |       3       |   r  d2     |
                    ---------------------------------------------

                    ---------------------------------------------
                    |       Machine  2  State Transitions       |
                    ---------------------------------------------
                    | From  |  To  | other machine | Transition  |
                    ---------------------------------------------
                    |   1   |  2   |       1       |   r  d0     |
                    |   1   |  3   |       1       |   r  d3     |
                    |   2   |  1   |       3       |   s  d1     |
                    ---------------------------------------------

                    ---------------------------------------------
                    |       Machine  3  State Transitions       |
                    ---------------------------------------------
                    | From  |  To  | other machine | Transition  |
                    ---------------------------------------------
                    |   1   |  2   |       2       |   r  d1     |
                    |   2   |  1   |       1       |   s  d2     |
                    |   2   |  3   |       1       |   s  d4     |
                    ---------------------------------------------
                    REACHABILITY GRAPH
  1 [ 1,E,E, 1,E,E, 1,E,E]
       -d0  2  [ 2,d0 ,E, 1,E,E, 1,E,E]   2
       -d3  2  [ 3,d3 ,E, 1,E,E, 1,E,E]   3
  2 [ 2,d0 ,E, 1,E,E, 1,E,E]
       +d0  1  [ 2,E,E, 2,E,E, 1,E,E]   4
  3 [ 3,d3 ,E, 1,E,E, 1,E,E]
       +d3  1  [ 3,E,E, 3,E,E, 1,E,E]   5
  4 [ 2,E,E, 2,E,E, 1,E,E]
       -d1  3  [ 2,E,E, 1,E,d1 , 1,E,E]   6
  5 [ 3,E,E, 3,E,E, 1,E,E]**********DEADLOCK condition**************
  6 [ 2,E,E, 1,E,d1 , 1,E,E]
       +d1  2  [ 2,E,E, 1,E,E, 2,E,E]   7
  7 [ 2,E,E, 1,E,E, 2,E,E]
       -d2  1  [ 2,E,E, 1,E,E, 1,d2 ,E]   8
       -d4  1  [ 2,E,E, 1,E,E, 3,d4 ,E]   9
  8 [ 2,E,E, 1,E,E, 1,d2 ,E]
       +d2  3  [ 1,E,E, 1,E,E, 1,E,E]   0
  9 [ 2,E,E, 1,E,E, 3,d4 ,E]**********Unspecified Reception***********

SUMMARY OF REACHABILITY ANALYSIS (ANALYSIS COMPLETED)
-----------------------------------------------------
Total number of states generated : 9
Number of states analyzed : 9
number of deadlocks : 1
number of unspecified receptions : 1
maximum message queue size :  1
channel overflow :NONE

                                UNEXECUTED TRANSITIONS
                                  *****NONE*****
```

Figure 16 : Program Output for the example protocol

compiles them with the other compilation units, the analysis of the specified protocol can be performed. Construction of the specification in the form of Ada packages and subprograms is explained in the next section.

TABLE 3: BIG MUSHROOM PROGRAM COMPILATION UNITS

Compilation Unit	Description	File Name
main(procedure)	This is the *parent unit*. Contains the main data structures, global variable and the driver.	smain.a
load_machine_array (procedure)	Builds the adjacency lists from FSMs.	sinput.a
read_in_file(procedure)	Parses the input FSM text file	sinput.a
build_Gstate_graph (procedure)	Generates the global reachability graph.	sg_treachability.a
Global_hash(function)	Generates an index number according to the SuperTrace hashing function for Big mushroom.	sg_reachability.a
hash(function)	Generates an index number according to the hashing function for Smart mushroom option.	sg_reachabilty.a
clear_pointers(procedure)	Deallocates the dynamic memory space for another analysis	sg_reachabilty.a
search_for_Stuple (function)	Searchs the reachability graph for the equivalent system tuples using hashing	sg_search.a
clear_hs_hash_array (procedure)	Clears the hash array and deallocates the memory for system state analysis	sg_search.a
output_Gstate_node (procedure)	Outputs the machine states, and the states with deadlocks for global reachability analysis.	sg_output.a
output_sys_node (procedure)	Outputs machine states, and states with deadlocks for system state analysis.	sg_output.a
output_Gstate_transition (procedure)	Outputs the transition name for global reachability analysis.	sg_output.a
output_sys_transition (procedure)	Outputs the transition name for system state analysis	sg_output
output_unexecuted_transitions (procedure)	Outputs the unexecuted transitions	sg_output.a
output_machine_arrays	Outputs the FSM description in a tabular format	sg_output.a
output_analysis(procedure)	Driver for the output subprograms	sg_output.a
create_output_file (procedure)	Creates an output file for storing the analysis results	sg_output.a
system_call(procedure)	Interface procedure for Unix system calls via C.	ssystem.a
queues(generic package)	Implements the queue operations for the pointer queue that stores the nodes temporarily.	squeues.a
stacks (generic package)	Implements the stack operation for storing enabled transition	sstacks.a
definitions (package)	Includes user defined local and shared variables	named by the user
Analyze_Predicates (procedure) there is one for each machine	Determines the enabled transitions from the predicates	named by the user

TABLE 3: BIG MUSHROOM PROGRAM COMPILATION UNITS

Compilation Unit	Description	File Name
Action (procedure)	Executes the actions for the enabled transitions.	named by the user
output_gtuple (procedure)	Outputs the global state tuples in a format defined by the user.	named by the user

2. Input

The inputs to the program consists of three parts, as mentioned earlier. FSMs are entered using a text file representation as in *Simple Mushroom* program. Variables and predicate-action table are entered as Ada packages/subprograms. The user needs to complete these packages and subprograms by filling in templates provided.

The Ada package template for the variable declarations is called "definitions." The predicate-action table is entered using an Ada subprogram template which consists of one procedure named "Action" and two to eight procedures called "Analyze_Predicate_Machine*" according to the number of machines in the protocol. The "*" at the end of the procedure name is replaced by the corresponding machine number for each machine in the protocol.

After completing the templates described above, the user must compile these units with the other compilation units listed in Table 3. Since the completion of these was explained in [BULB93], they will not described here. But our example protocol *stop-and-wait* in Chapter II is used to illustrate how to complete the templates.

a. Finite State Machines

There are a few differences in the FSM description of *Big Mushroom* program from *Simple Mushroom* program. In the SCM model, explicit machine numbers to show which machine the message sent to or received from are not needed for the transition names. Since shared variables are used for communication between machines, this information is included in the predicate-action table. The FSM text file for the example ring protocol is shown in Figure 17.

The FSM text file is read by the input procedures and the adjacency list, which is used during the construction of system and global reachability graph is generated.

b. Variable Definitions

The user defines the protocol variables in Ada package named *definitions*. This package includes the local variables for each machine and the global variables, which are considered shared and allow communication between machines. A variable can be one of the Ada defined types such as: integer, array, string, record, character, boolean etc. These types and their

```
start
number_of_machines 3
machine 1
state 0
trans Snd_data 2
state 1
trans Rcv_Ack 2
machine 2
state 0
trans Rcv_data
state 1
trans Snd_Ack
initial_state 0 0
finish
```

Figure 17 : Text file description of the example ring protocol

subtypes are used to define the protocol variables. The variable declaration for the stop-and-wait protocol is shown in Figure 18.

c. Predicate-Action Table

The predicate-action table is represented by a number of subprograms as separate compilation units. These subprograms are named *Analyze-Predicates* and are used to determine the enabled transitions for each machine. The procedure named *Action* executes the actions to be taken for the corresponding enabled predicates. There is one *Analyze_Predicates* procedure for each machine and one *Action* procedure for the protocol.The user completes the template for each state of the machines. The predicate-action file for the example *stop-and-wait* protocol is shown in Figure 19.

The enabled transitions are passed into this procedure through the "in_transition" formal parameter and the necessary changes are made to the local and shared variables by the *Action* procedure. The "out_system_state"parameter passes the changed protocol variables to the calling procedure. The completed Action procedure is shown in Figure 20. Text in boldface shows the user defined parts.

```
with TEXT_IO;
use TEXT_IO;
  package definitions is
  num_of_machines : constant := 2;
  type acm_transition_type is (Snd_data, Rcv_data, Snd_Ack, Rcv_Ack, unused);

  type buffer_type is (D,A,E);
  package buff_enum_io is new enumeration_io (buffer_type);
  use buff_enum_io;
  type dummy_type is range 1..255;

  type machine1_state_type is
  record
    out_buff  : buffer_type := D;
  end record;
  type machine2_state_type is
  record
    in_buff : buffer_type := E;
  end record;
  type machine3_state_type is
  record
    dummy : dummy_type;
  end record;
                                                    .
                                                    .
                                                    .
                                                    .
  type machine8_state_type is
  record
    dummy : dummy_type;
  end record;
  the global_variable_type is
  record
    CHAN : buffer_type := E;
    RET  : buffer_type := E;
  end record;

end definitions;
```

Figure 18 : Completed *Definitions* package for *stop-and-wait* protocol

```
separate (main)
procedure Analyze_Predicates_Machine1(local : machine1_state_type;
                          GLOBAL:   global_variable_type;
                          s    : natural ;  w : in out transition_stack_package.stack) is
begin
  case s is
    when 0 =>
      if ((GLOBAL.CHAN = E) and (LOCAL.out_buff /= E)) then
        Push(w,Snd_data);
      end if;
    when 1 =>
      if (GLOBAL.RET = A) then
        Push(w,Rcv_Ack);
      end if;
    when others =>
      null;
  end case;
end Analyze_Predicates_Machine1;

procedure Analyze_Predicates_Machine2(local : machine2_state_type;
                          GLOBAL: global_variable_type;
                          s    : natural ;  w : in out transition_stack_package.stack) is
begin
  case s is
    when 0 =>
      if (GLOBAL.CHAN /= E) then
        Push(w,Rcv_data);
      end if;
    when 1 =>
      if true then
        Push(w,Snd_Ack);
      end if;
    when others =>
      null;
  end case;
end Analyze_Predicates_Machine2;

procedure Analyze_Predicates_Machine3(local : machine3_state_type;
                          GLOBAL: global_variable_type;
                          s    : natural ;  w : in out transition_stack_package.stack) is
begin
 null;
end Analyze_Predicates_Machine3;
                                          .
                                          .
                                          .
procedure Analyze_Predicates_Machine8(local : machine8_state_type;
                          GLOBAL: global_variable_type;
                          s    : natural ;  w : in out transition_stack_package.stack) is
begin
 null;
end Analyze_Predicates_Machine8;
```

Figure 19 : Completed *Analyze_Predicates* procedures for the *Stop-and-wait* protocol

```
separate (main)
procedure Action(in_system_state: in out Gstate_record_type;
                 in_transition : in out scm_transition_type;
                 out_system_state : in_out Gstate_record_type ) is
begin
  case (in_transition) is
  when (Snd_data) =>
    out_system_state.GLOBAL_VARIABLES.CHAN:= in_system_state.machine1_state.out_buff;
     out_system_state.machine1_state.out_buff := E;
  when (Rev_data) =>
    out_system_state.machine2_state.in_buff := in_system_state.GLOBAL_VARIABLES.CHAN;
  when (Snd_Ack) =>
    out_system_state.GLOBAL_VARIABLES.RET := A;
    out_system_state.machine2_state.in_buff := E;
  when (Rev_Ack) =>
    out_system_state.GLOBAL_VARIABLES.CHAN:= E;
    out_system_state.GLOBAL_VARIABLES.RET := E;
  when others => put_line("There is an error in the Action procedure");
  end case;
end Action
```

Figure 20 : Completed *Action* procedure for the *Stop-and-Wait* protocol

3. Global Reachability Analysis

The process of generating and examining the set of all reachable states from the initial state is called reachability analysis. The program is capable of generating both the global and system reachability analyses separately for a protocol formally specified by the SCM model. Since the system reachability analysis generates relatively small number of states Supertrace Algorithm is not used for that analysis.

The user can select either *global reachability analysis* or *system state analysis* from a menu. During the graph construction, the program also detects any deadlock conditions. Analysis results are stored in an output file named "rgraph.dat" in parallel with graph construction.

The structure of the global state used for the program is shown in Figure 21. This node structure also includes outgoing transitions. The maximum number of outgoing transitions is artificially limited to 7. It can be increased as necessary. The shared variables are stored in the "*global_variables*" variable and local variables are stored separately for each machine in "*machine_state**" variables.

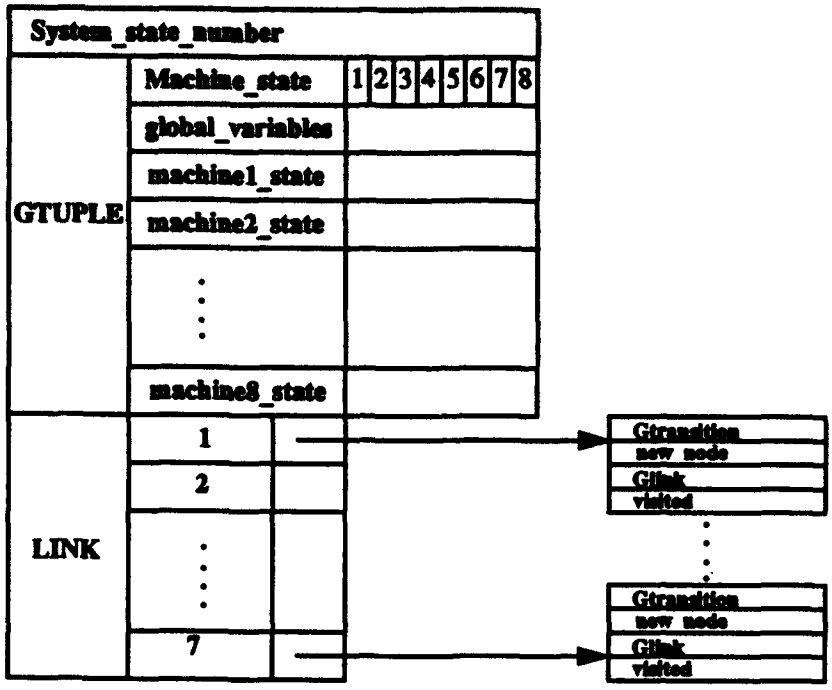

Figure 21 : Global State Structure with Outgoing Transitions

The initial global state is created from both the FSM text file and the initial values of the variables assigned in the definitions package. All the outgoing transitions are initially set to *null*. Starting with the initial global state, new nodes are added and linked to the graph. The pseudo-code algorithm for constructing the global reachability graph is shown in Figure 22.

The program implements hashing to search through hash table for duplicate states which increases the run time efficiency of the analysis. There is a major difference between the *Simple mushroom* and the *Big mushroom* hashing functions. In the *Simple mushroom program* the user does not need to specify a hashing function. A predetermined function which considers machine states and message queues is implemented in the program. For the *Big mushroom program* the user must design and enter a global hashing function. The function must account for machine states, local, and global variables. An example of a global hash function for Stop-and-wait protocol is given in Figure 23.

4. Output

The program stores the results of the analysis in a file named "rgraph.dat." This file contains FSMs in a tabular format, system/global reachability graph, and the results of analysis

```
loop(main loop)
  for index1 in 1 .. total_number_of_machines loop
    position_holder(index1) := machine_array(index1)(M_state(index1))
    Determine the enabled transitions for the machine(index1) and push into transition_stack
    While not Empty(transition_stack) loop
      while (position_holder(index1) /= null) loop
        Traverse the machine arrays for each enabled transition in the stack
        if a transition found in the machine arrays
          create a temporary node resulting from this transition
          call Action procedure to make the necessary changes to the variables of this node
          Search the Hash look-up table to see this node was created(redundant)
          If the table slot corresponding to the index created by hash function is not set(false) then
            set the table slot(true)
            Enqueue the node into the Gpointer_queue
          else
            write transition to the output file and discard the node
          end if
        else
          position_holder(index1) := position_holder(index1).Slink
        end if
      end loop
      if not Empty(transition_stack) and a transition not found in the machine arrays
        pop the stack
      end if
    end loop
  end loop
  If Gpointer_queue Empty then
    exit
  else
    Dequeue Gpointer_queue
    Update Mstate for this new node
  end if
end loop (main loop)
```

Figure 22 : Algorithm for Generating Global Reachability Graph for *Big Mushroom*

```
function GLOBAL_HASH ( current_gstate : Gstate_record_type) return integer is
  index: integer:=0;
  sum:integer:=0;
  m : machine_state_array := current_gstate.machine_state;
begin
  index :=  ( (m(8) *83999) + ( m(7) * 72888) + (m(6) *61997) + (m(5) *5995) +
              (m(4) * 46571) +(m(3) * 34677) + (m(2) * 21323) + (m(1) *18203) ) ;
  sum := buffer_type'pos(current_gstate.machine1_state.out_buff)*373351+
         buffer_type'pos(current_gstate.machine2_state.in_buff)*677139+
         buffer_type'pos(current_gstate.GLOBAL_VARIABLES.CHAN)*973551+
         buffer_type'pos(current_gstate.GLOBAL_VARIABLES.RET)*123551;
  return ((index*3+sum*7) mod 1545423);
  end GLOBAL_HASH;
```

Figure 23 : Global Hash function for *Stop-and-wait* protocol

consisting of number of states generated, number of states analyzed, and number of deadlocks. Unexecuted transitions are also listed at the end of the analysis.

Since each protocol specification has different variables, the user also has the flexibility to output the desired variables. This is done in a similar manner to the predicate-action table and

variable definitions representation explained in [BULB93] using an Ada procedure template.The user completes the template with Ada "put" statements for outputting the global states. Since the system state tuples do not include the variables, there is no need to define an output format for system reachability graph. The completed template for the output_Gtuple procedure for stop-and-wait protocol is also given in Figure 24.

```
separate (main)
procedure output_Gtuple (tuple : in out Gstate_record_type) is
begin
   if print_header then
      new_line(2);
      set_col(5);
      put_line (" m1(out_buff),m2(in_buff), (CHAN, RET)");
      print_header := false;
   else
      put(" ["&integer'image(tuple.machine_state(1)) );
      put(",");
      buff_enum_io.put(tuple.machine1_state.out_buff);
      put(",");
      put(" ["&integer'image(tuple.machine_state(2)) );
      put(",");
      buff_enum_io.put(tuple.machine1_state.in_buff);
      put(",");
      buff_enum_io.put(tuple.GLOBAL_VARIABLES.CHAN);
      put(",");
      buff_enum_io.put(tuple.GLOBAL_VARIABLES.RET);
      put("]");
   end if;
end output_Gtuple;
```

Figure 24 : Completed output_Gtuple procedure for Stop-and-wait protocol

The output of the program for the example ring protocol is given in Figure 25.

```
REACHABILITY ANALYSIS of :stopwait.scm
                SPECIFICATION

-------------------------------
| Machine 1 State Transitions |
-------------------------------
| From  |  To  |  Transition  |
-------------------------------
|   0   |   1  |  snd_data    |
|   1   |   0  |  rcv_ack     |
-------------------------------

-------------------------------
| Machine 2 State Transitions |
-------------------------------
| From  |  To  |  Transition  |
-------------------------------
|   0   |   1  |  rcv_data    |
|   1   |   0  |  snd_ack     |
-------------------------------
 REACHABILITY GRAPH

   m1, out_buff ,m2 , in_buff, CHAN, RET

     0  [ 0 , D ,  0 , E , E , E ]                  snd_data   1
     1  [ 1 , D ,  0 , E , D , E ]                  rcv_data   2
     2  [ 1 , D ,  1 , D , D , E ]                  snd_ack    3
     3  [ 1 , D ,  0 , D , D , A ]                  rcv_ack    0

SUMMARY OF REACHABILITY ANALYSIS (ANALYSIS COMPLETED)
-----------------------------------------------
Number of states generated :4
Number of states analyzed :4
Number of deadlocks : 0

                        UNEXECUTED TRANSITIONS
                            *****NONE*****
```

Figure 25 : The Output of the Program for the Example Ring Protocol

D. Summary

In this chapter, example protocols in Chapter II were analyzed to demonstrate the usage of Mushroom program. The protocols analyzed in this chapter are intentionally chosen simple to help the user understand the mushroom program's inputs and outputs. However, the analysis results verifies that Supertrace algorithm approximates the full search method by generating the same outputs obtained manually in Chapter II. The major achievement of Supertrace will be illustrated in Chapter V with larger protocols.

IV. A PROGRAM FOR PROTOCOL TEST SEQUENCE GENERATION

In this chapter, the concept of conformance testing is first introduced; next, a procedure created for test sequence generation [LUND90A] is discussed. Finally, "TESTGEN," the program which automates the test sequence generation is illustrated.

A. Introduction To Conformance Testing

A conformance test is used to ensure that the external behavior of an implementation of a protocol is equivalent to its formal specification. In conducting a conformance test we are given a known protocol specification and an unknown implementation. The implementation, for practical purposes, is considered as a black box with a finite set of inputs and outputs. The test provides a sequence of input signals, and observes the resulting outputs. The *implementation under test* (IUT) should pass the test only if all observed outputs match those prescribed by the formal specification. The series of input sequences which are used to exercise the protocol implementation in this way are referred as *conformance test sequence* throughout this thesis.

Two problems with conformance testing need to be solved:

1. Find a general, applicable, efficient procedure for generating a conformance test sequence for a given protocol implementation, and

2. Find a method for applying the test sequence to a running implementation.

This first issue is the focus of this thesis while the second problem is beyond the scope of this thesis.

It is desirable to have the specification of a protocol expressed in a formal model and the specification formally verified.

A previous study [MILL90] on this issue observed gaps between the specification, the verification, and the conformance testing of network protocols. Protocol models which are designed for specification purposes usually have many powerful program language constructs, to simplify the specification, but are difficult to analyze. Protocol models designed primarily for analysis purposes, such as the CFSM model, are often too simple for the specification of modern, complex protocols. Much recent work on conformance testing starts from the description of a protocol as an incompletely specified finite state machine with input/output labels on the transitions[CHEN90][DAHB90]. Normally protocol specifications are not described in this manner.

Suppose a test designer was to design a test for a protocol specified using the formal language LOTOS. First, he must translate the specification to an I/O diagram. This is a difficult and complex

process, and during which errors are easily introduced. Only then, when this translation is complete, can he begin to generate the tests for conformance testing.

The automation of the test sequence generation [LUND90A] is an attempt to close the gap between specification/verification and testing of protocols. In this thesis, the test generation starts from a protocol model, designed for the specification and verification of protocols. A procedure created in [LUND90B], is used for the generation of a test sequence for a protocol specified in the SCM model. This procedure and its automation as a software tool does not guarantee that all the errors or combination of errors in a protocol are found. But they do represent an attempt to exercise all parts of protocols providing some assurance that the implementation meets its purpose.

B. Test Generation Procedure

In this section a procedure and its automation are described for generating a sequence of tests for a protocol specified as a SCM model. The input is the formal protocol specification (FSM and predicate-action table) specified as a *system of communicating machines* (SCM). The output is a sequence of tests and an I/O diagram in a tabular format. The generated sequence is intended to be applied to an IUT.

The sample IUT throughout this section is the network node for CSMA/CD protocol. Before generating the sequence of tests and the I/O diagram for each test in the sequence, shared and local variables must be identified. The test inputs (the shared and local variables that can be set in a controlled way) and the outputs (the shared and local variables can be observed for test purposes) should be identified. These inputs and outputs form the I/O for the test steps.

The format for each single test is

$S_I\ i_1, i_2, \ldots, i_n\ ;\ o_1, o_2, \ldots, o_m\ S_E$

S_I is the state of machine when the test begins. The $i_1, i_2, \ldots, i_n$ are the input values at the start of test execution. The $o_1, o_2, \ldots, o_m$ are the values of the output variables after test execution. S_E is the state of the machine when the test is complete. The input and the output variables are taken from the shared and local variables of the machine. The determination of these variables is explained in the following section.

The procedure explained in the following sections is taken from [LUND90A]. It is written in three parts:

- Preliminary steps,
- Test sequence generating procedure, and
- Refining steps.

1. Preliminary Steps

1. From the machine specification FSM diagram, mark each transition whose name appears on more than one transition. Each such instance for a given name is given a separate distinguishing label.

2. From the *predicate-action table*, note the number of clauses in each enabling predicate. Mark each clause. An enabling predicate may consist of several clauses, any one of which might be true, allowing the transition to execute. Marking each clause insures that each one is tested individually.

3. For each shared variable x, determine if x is an input variable, an output variable, or both. For each x which is both, split x into two variables, x_i and x_o for testing purposes.

4. For each local variable l, determine if l is used as an interface to the higher layer user of this protocol. If so mark l as input, output or both. Each such local variable is specifically designated, and is an input variable if it appears in an enabling predicate, and an output variable if it appears in an Action part of *predicate-action table*. If l is both input and output, split it into two variables l_i and l_o for test purposes.

2. Test Sequence Generating Procedure

Initially the test sequence is empty.

1. *state* $\leftarrow$ initial state.

2. Let $t = (p,a)$ be an untested transition from *state*.

(a) Determine the values of the input variables which make exactly one of the untested clauses of p true. Check to see if these values allow any other transition from this state to be executed. If there is one, set additional input variables to values that insure only the transition under test is enabled. Fill these in, and mark others "DC" for "don't care."

(b) Determine and mark the expected values for the output variables; also record the expected values assumed by the local variables.

(c) Set S_I to state; determine the next state and set S_E to it.

(d) Determine if S_E is transient; if not mark it as a "stop state" and skip to (3). The state is *transient* if one of its enabling predicates is true immediately upon reaching the state. This means that it can pass on to another state immediately, without waiting for further input.

(e) Attempt to make S_E into a stop state by setting "DC" values. That is, make the DC values such that, upon reaching state S_E, none of the enabling predicates are true. If successful, go to (3).

(f) If S_E is a transient state and more than one transition leaving S_E is enabled, choose one and set inputs not yet specified (if any exist), so that only one transition leaving S_E is enabled; set $t = (p,a)$ to this transition.

3. Output this test $S_I\, i_1, i_2, \ldots, i_n / o_1, o_2, \ldots, o_m\, S_E$ as the next test in the test sequence.

4. Mark the clause just tested. If all clauses in transition t are now tested, mark t as tested. If all transitions are now marked as tested, exit to "refining steps." Otherwise, continue to step (5).

5. Set state to S_E. If state is a stop state go to (2), otherwise go to step2(b).

Step 2(a) assumes that it is possible to set the input variables to values that make exactly one of the clauses true. If the protocol is well designed this assumption will generally be true. However, there is always a possibility this is not the case; if so, the test designer must choose the values so that the clauses will be tested as thoroughly as possible, perhaps in combination with other clauses. If a clause cannot be tested individually, the question of its necessity to the specification should be considered.

Step 5 sets the starting state of the next test in the sequence to the ending state of the current test. This makes the ordering of the tests follow the order of their occurrence in the actual protocol execution.

3. Refining Steps

1. Construct the I/O state diagram from the test sequence.

2. Determine if the sequence are unique, so that from each state, we have a unique input output (UIO) sequence to confirm. If not attempt to extend the sequence so that we have a unique UIO sequence from each state.

3. Check for any converging transitions. Mark these, as potential problems for testing.

The I/O diagram can be constructed from the test sequence and is a tool to help the test designer insure completeness. This finite state machine is often used as the starting point in test generation in the literature.

A UIO sequence has been defined as a sequence of inputs such that, if the input sequence is applied to the FSM when FSM is in state i, the resulting output sequence could not have been produced by the FSM when the FSM is in any other state [DAHB90][SIDH88]. If the sequence of tests applied to a machine implementation in a state i is a UIO sequence, and the output is expected, then we have a stronger argument that the machine was, in fact, in state i.

C. Test Generation of the CSMA/CD Protocol

In this section, the test generation procedure is illustrated through an application on a well known protocol for local area networks, the CSMA/CD (carrier sense multiple access with collision detection) protocol. The protocol has a formal specification as a SCM model in [LUND93].

The topology of the CSMA/CD is a simple bus with a single channel, as in displayed in Figure 26. All stations transmit and receive on the channel. If more than one station transmits simultaneously, interference or "collision" occurs. A station wishing to transmit first checks the medium. If no other transmission is detected, it begins transmitting its own message. If a collision occurs, the station attempts to retransmit its message after waiting a random time period.

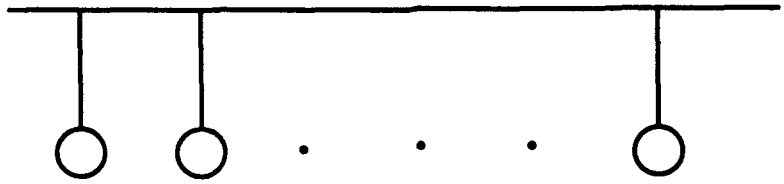

Figure 26 : Topology of the CSMA/CD Network

The specification of CSMA/CD protocol consists of the finite state machine and the local variables of the network stations (Figure 27) and the predicate action table for the network stations (Table 4). The shared variables, *Medium* and *Signal* and finite state machine of the controller, responsible for the control of shared variables, are shown in Figure 28.

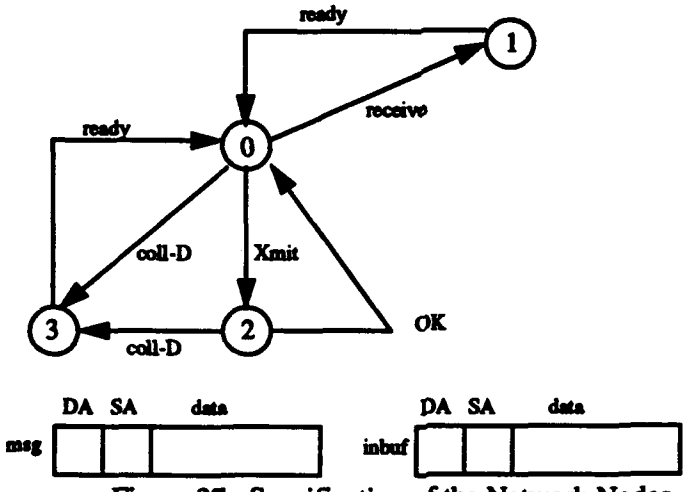

Figure 27 : Specification of the Network Nodes

The predicate action table of Controller is shown in Table 5.

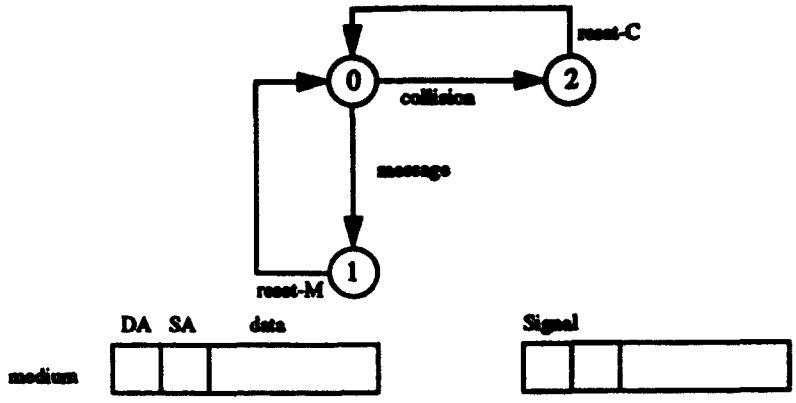

Figure 28 : Controller and Shared Variables

The local variables of each network node are *msg* and *inbuf*. *Msg* is of the same type as *medium*. *Inbuf* is used to receive incoming messages. State 0 is the initial state, from which either a *receive* or *transmit* action is initiated. States 0,2, and 3 make up the transmit/collision states, and states 0 and 1 comprise the receiving portion of the machine.

The controller continually monitors the communication medium. Whenever a nonempty value is detected it transitions to either state 2 or 1, according to whether a collision or good transmission occurred. If a collision occurs (*medium = undefined*), the controller moves to state 2. When all stations have detected the collision (*Signal(1..n) = collision*), the controller clears the medium and returns to 0. If a good transmission occurs, the controller moves to state 1. After receiving station accepts the message, the controller clears the *medium* and returns to 0. The predicate-action table for controller is shown in Table 5.

The network stations may either transmit or receive from the initial state 0. If a station, in state 0 has data to transmit, indicated by a nonempty *msg*, and the *medium* is clear, it will transition to state 2 and the message written to *medium*. The variable *msg* becomes nonempty when the upper layer of the protocol has data to send. If no collision occurred the *OK* transition will set the state back to 0. This is indicated by the value of *Signal(i)*, being set to *clear* by the controller, providing if no collisions occurred. If a collision occurs, then the *coll-D (collision detected)* transition will be taken. Once the controller clear the *medium*, indicated by *Signal(i):= clear*, the node will return to state 0 and attempt to retransmit.

The *receive* transition is also starts from state *0*. This transition becomes enabled when a message appears in *medium* with the station's address in *medium.DA*. The node copies the message into its input buffer *inbuf*, then signals the controller by setting *Signal(i)* to *transceive* and returns to state *0*.

TABLE 4: PREDICATE ACTION TABLE FOR NETWORK NODES

Transition	Predicate	Action
Xmit	msg $\neq \varnothing \wedge$ medium $= \varnothing$	medium := msg; Signal(i) := transceive
OK	Signal(i) = clear	msg := $\varnothing$
coll-D	medium = undefined	Signal(i) := collision
ready	Signal(i) = clear	
receive	medium.DA	inbuf := medium; Signal(i) := transceive

Generation of the Protocol test sequence will be discussed later in this chapter along with the software tool TESTGEN.

TABLE 5: PREDICATE-ACTION TABLE FOR THE CONTROLLER

Transition	Predicate	Action
message	$\neg$medium $\in$ {undefined,$\varnothing$}	
reset-M	Signal(medium.DA) = transceive	medium := $\varnothing$; Signal(1..n) := clear
collision	medium = undefined	
reset-C	Signal(1..n) = collusion	medium := $\varnothing$; Signal(1..n) := clear

1. Creating Inputs For The "TESTGEN" Program

The software tool that automates the generation of test sequences is called "TESTGEN." The general structure of TESTGEN is shown in Figure 29. The inputs of the program are two text files which are created and named by the user.

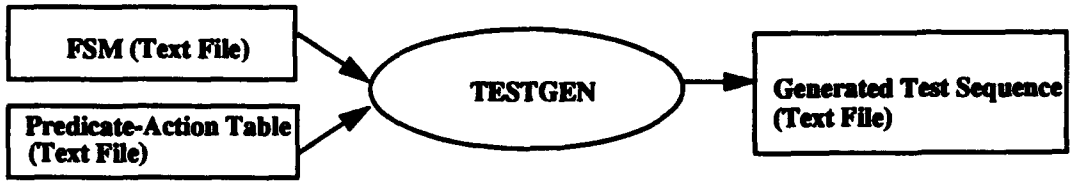

Figure 29 : The General Structure of TESTGEN Program

The input files are easily created utilizing the following procedures. Before creating the FSM input file, the user should assign a number to each transition of the FSM. This distinguishes each arc, even though they may represent the same transition name. The numbered FSM of the CSMA/CD protocol is shown in Figure 30.

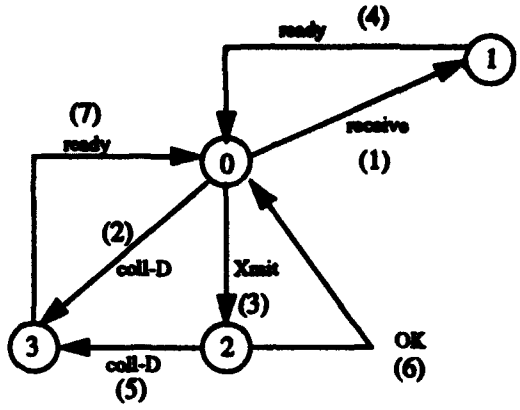

Figure 30 : Assignment of Numbers to Transitions of CSMA/CD Protocol

To create the first file, the user first specifies the initial state of the FSM as the first line in the FSM input file. Each line, thereafter, represents a transition arc and is entered in the format

From State *To State* *Number Assigned* *Transition Name*

with a single space between each field.

It is a practical way to enter transition arcs starting from initial state, listing all outgoing arcs and then continuing with the next state. Transition arcs can be entered in any order as long as they have the previous structure.

An example FSM input file for the CSMA/CD protocol is shown in Figure 31. The "0" in the first line shows the initial state of our example CSMA/CD protocol.

```
0
0 1 1 receive
0 2 3 xmit
0 3 2 coll-D
1 0 4 ready
2 0 4 ok
2 3 5 coll-D
3 0 7 ready
```

Figure 31 : FSM Input File of CSMA/CD Protocol

Figure 32 shows the parts of a transition arc and their meanings in FSM input file.

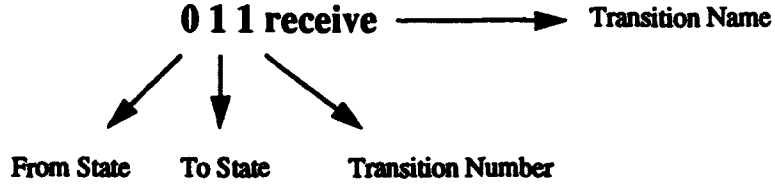

Figure 32 :Representation of Transition Arcs in FSM Input File.

The second input file contains *predicate action table(PAT)* of the specified protocol. This file is created in the same tabular format as the *predicate-action table*. Each column of the *PAT* is separated with vertical bar ' | ' with a space on each side, so that it is distinguishable from the other table entrees. The ' | ' delineates the borders of transition, predicate and action columns of the *PAT*. Multiple action statement should be separated with a semi-colon (;). If no action is to be taken for a transition, the keyword *"no"* must be entered as the action part of the input file. If a transition occurs every time we enter a state, it is indicated by putting keyword the *"true"* in the predicate part of the input file. An example of predicate-action input for the CSMA/CD protocol is shown in Figure 33.

```
xmit     | msg /= empty and medium = empty | medium := msg ; signal(i) := transceive  |
ok       | signal(i) = clear               | msg := empty                             |
coll-D   | medium = unidentif              | signal(i) := collision                   |
ready    | signal(i) = clear               | no                                       |
receive  | medium = (x,x,i)                | inbuf := medium ; signal(i) := transceive |
```

Figure 33 : Predicate-Action File Input of CSMA/CD Protocol

An example line in the predicate-action input file is shown in Figure 34.

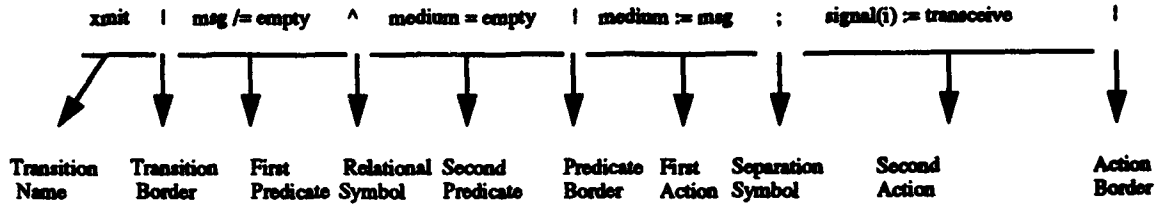

Figure 34 :Example Input line of Predicate-Action File

Since the predicate action input is a text file, some relational symbols are not readily apparent. They need to be represented in a format that can be easily entered from the keyboard yet understandable by the program. The method used in this thesis to handle this problem is shown in Figure 35.

If there is more than one clause in a disjunctive predicate part of a transition it is difficult to determine which predicates need to be enabled to make a transition occur. The TESTGEN program is capable of parsing and presenting clauses in following form

- first clause relational symbol second clause
- first clause relational symbol (second clause relational symbol third clause)
- (first clause relational symbol second clause) relational symbol third clause

The TESTGEN program represent these relational clauses by putting the relational symbol between two clauses together with the values of the input variable to the output table. The relational symbol between the relational clauses in parenthesis is put in the output file in parenthesis

Relational Symbols	→	Text Symbols
x ≠ y	→	x/=y
∅	→	empty
x ∧ y	→	x and y
x ∨ y	→	x or y
'Assignment to a variable	→	:=
x ⊕ y	→	x mod+ y

Figure 35 : Relational Symbols and Their Representations

so it is distinguishable from other relational symbol. If the enabling predicate has more than three clauses the TESTGEN program may not correctly represent these clauses in the output test sequence. The user should control the output test sequence for these transitions.

If input variables are record structures such as *medium*, *msg*, *inbuf*, assignment or comparison of a specific fields of the record are done within parentheses and by putting "x" in the positions that is unimportant. For example, assume a variable "Z" is a record structure with three subparts a, b and c. Assignment of the value "3" to the 'a' field of Z should be in the format "*Z:= (3,x,x).*" This means 3 is assigned to 'a' and no changes are made to 'b' and 'c.' The TESTGEN program finds local and shared variable by parsing predicate action input file so instead of entering different representations of one variable such as *medium.DA* or *medium.SA*, entering variables in this format helps program determine the variable structure and makes output file easy to read.

Comparisons and assignments to arrays should be entered in the format *A(i)=value*. This may create more than one representation of the same variable in the output file but it makes the output test sequence more understandable.

2. Procedure Of The Protocol Test Sequence Generator

The algorithm of the test generator consists of two major subparts: the first part finds all possible paths and cycles in the FSM starting from the initial state. It prints the list of paths and cycles to a text output file, named by the user. It also ensures that there is a path from all cycles eventually returning to the start state. If it can't find such a path it will print out a message, warning the user of possible errors in the specification of the protocol.The pseudo-code algorithm for finding

all paths and cycles of FSM is illustrated in Figure 36. Finding all possible transition sequences ensures that each instance of each transition is tested.

```
Parse the FSM input file and make a list of transition arcs(list_of_transitions);
Take one arc originating from the initial state put it into a list_of_paths;
If there is more than one arc
  Append other arcs to the end of list_of_paths
end if;
Start with the first arc in the list_of_paths and find the destination node
Main loop:
loop until there is no path processed in the list_of_paths
  Look for other arcs originating from the destination node in the list_of_transition
  If there is one;
    Check that arc is put in the path generated
    if it is
      Mark the path as cycle found
      Mark the path generated as processed and skip the next path in the list_of_paths
      replace the starting arc with the arc at the end of the path on the next unprocessed path
      go to the main loop
    else
      Append that arc to the original arc
    end if;
  elsif there is more than one arc
    Copy the path generated and append the copy to the end of list_of_paths along
    with the other arc or arcs originating from destination node appended
  else
    "There may be an error in the protocol. Inform the user."
  end if;
  check to see destination node is initial state
  if it is then
    mark the path generated as a new path and skip to the next path in the list_of_paths
    replace the starting arc with the arc at the end of the path on the next unprocessed path
  else
    replace the starting arc with the arc originating from the destination node
  end if;
end loop;
```

Figure 36 : Algorithm for Finding Paths and Cycles in the FSM

To trace all the possible paths which could be generated, a queue of linked lists is implemented. The trace is as follows: Starting with the initial state, all transitions are placed into the queue. The first entry is dequeued, becoming the current entry, and is used to continue the trace. The current entry remains so until it describes a cycle back to the initial state.

All transitions out of the last node of the current path are determined, and one of them is appended to the current entry.

Any other transitions are each appended to a copy of the current path and placed at the end of the queue (list_of_paths). When the initial state is reached, next path in the queue becomes current path. This procedure continues until the queue is empty.

The program starts with an arc originating from the initial state. In our example CSMA/ CD protocol the first arc selected is transition #1 (0 1 1 receive). It is inserted to the list_of_paths.

Since there is more than one transition leaving the initial state, the other (0 2 3 transmit), (0 3 2 coll-D) arcs are also inserted to the list_of_paths. Then destination node "1" of transition #1 is found from the list_of_transition and since there is one transition (transition #4) leaving destination node; it is appended to the end of our path. Then transition #4 becomes current arc. Since the destination node of the transition #4 is 0 (initial state) the path is marked as processed. The current entry becomes the last arc in the next unprocessed transition sequence (transition #3). The procedure continues until all paths and cycles originating from the initial state are found. The steps of finding paths and final path list at the end of procedure FIND_PATHS for CSMA/CD protocol is shown in Figure 37.

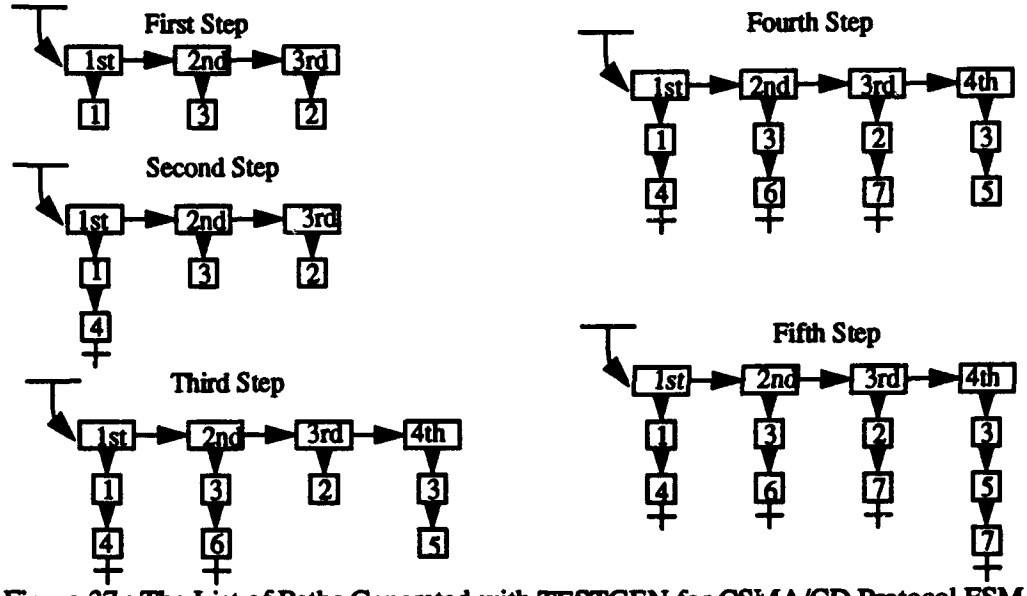

Figure 37 : The List of Paths Generated with TESTGEN for CSMA/CD Protocol FSM

3. Preliminaries

In our example many of our variables perform as both input and output sources. The shared variables *medium*, *Signal* and local variable *msg* are input and output variables. The second part of the TESTGEN determines our input and output variables. If a variable is used as both an input and output variable it is marked by placing *(i)* or *(o)* next to them to indicate its current usage.The program reads the transitions, predicates and actions associated with each transition from the *predicate action table* (PAT). It then creates the test sequence table and lists all transition sequences starting from the initial state by using *list_of_paths*. It prints each transition with the expected values of any local and shared variables. It also prints the action to be taken if the predicate

associated with transition is enabled. Pseudo-code of the second part of TESTGEN is shown in Figure 38.

```
Parse the predicate action input file
Determine transitions, local and shared variables predicates and actions associated with each
transition
Determine and mark the expected values for the output variables and record the expected values
assumed by local variables for each transition
Print the input , output, and shared variables
Take the first path from the list_of_paths
loop until no more list remained in the list_of_files
  begin with the first transition in the path
  set Si to the originating node of the transition
  set input variables of this transition according to the predicate action table
  if input variable is a record type
    set unimportant fields with "x"
  end if;
  set other input variables "DC" for don't care
  set output variables
  set Se to the terminal state of current transition
  Print the completed test to the output file
  set Si to the Se
  if not end of path
    replace the current transition with the next transition in the path
  else
    mark the path as processed
    replace the current transition with the first transition of the next unprocessed path
  end if;
end loop
```

Figure 38 : Pseudo-Code Algorithm for Generating Protocol Test Sequence

4. Test Sequence Generation

The TESTGEN program begins with the first transition *(#1 receive)* in the path list generated by the FIND_PATHS procedure. According to the predicate action input file to enable this transition, the DA field of medium must be set to the station's address, which we assume to be i. The remaining fields of the record *medium* may be any values, and are indicated by 'x' in the output table (Figure 39). The other input variables are set to "*don't care*" or *DC*.

When the *receive* transition occurs, *signal(i)* should be set to *transceive*, and *inbuf* should contain the value which was previously in *medium*. S_I is set to *source state* of the current transition (in this case 0), and S_E to the to *terminal state* (in this case 1). This completes the first test in the sequence and these values are output.The clause and transition are now marked "tested". The value of S_I is now set to 1, and next transition in the path is called.

The next iteration is the *ready* transition from state 1. The values selected are the second test in the output table (Figure 39). The ending state of this test is state 0 the initial state, so the path is marked as processed.

At the next iteration first transition in the next unprocessed path (*xmit*) is chosen, followed by the *OK* transition back to state 0. The same process continues with transition *coll-D*, which takes the machine state 3, and the *ready* transition returns it to state 0. Then the *Xmit* transition is chosen a second time in the last path which takes the machine state 2; then transition *coll-D* is chosen which is different from previous sequence; that takes the machine to state 3 and *ready* transition again returns it to the initial state. At this point all possible transition sequences have been processed.

The table generated by the TESTGEN program for the CSMA/CD protocol is shown in Figure 39. The table lists all nine possible transitions according to their order of occurrence. It is relatively easy to test all sequences of a transitions by simply following the order in the table.

Trans		input variables			**	output variables				
name	Si	medium(i)	msg(i)	signal(i)	**	inbuf	medium(o)	msg(o)	signal(o)	Se
receive	0	(i,x,x)	DC	DC	**	medium	--	--	transceive	1
ready	1	DC	DC	clear	**	--	--	--	--	0
xmit	0	empty	/=empty	DC	**	--	msg	--	transceive	2
ok	2	DC	DC	clear	**	--	--	empty	--	0
coll-D	0	undefined	DC	DC	**	--	--	--	collision	3
ready	3	DC	DC	clear	**	--	--	--	--	0
xmit	0	empty	/=empty	DC	**	--	msg	--	transceive	2
coll-D	2	undefined	DC	DC	**	--	--	--	collision	3
ready	3	DC	DC	clear	**	--	--	--	--	0

Figure 39 : The Test Sequence Table Generated with TESTGEN for CSMA/CD protocol

5. Refinement

The first refining step calls for the construction of the I/O diagram. This diagram can be constructed from the sequence of tests generated. In this case, because there are no transient states, there are four states which correspond to the four states of the specification; and the arcs between states are the same set as in the specification. The only difference is in the labeling of the arcs; for the I/O diagram, the label on each arc is the set of values if the input and output variables, as shown in output table Figure 39.

Next we must determine if the sequence is a UIO sequence. Consider the first test in the table, the *receive* transition. If the machine is in state 0 and we apply the inputs for the first test, the outputs are the *transceive* value in *Signal(i)* and a copy of medium in inbuf. The user may confirm that in no other state does this combination occur; so for the first state and test, we have an UIO sequence. From state 1, the *ready* transition is considered. This transition leads back to state 0; note that another *ready* transition leads from state 3 to state 0. This means that there is not a UIO sequence for states 1 and 3. This makes it difficult for the test designer to confirm these states. There

is however a UIO sequence leading into these states; so the lack of a UIO sequence from these states is less disturbing.

Finally a check for converging transitions shows that there is one case of this: the *ready* transition, leading to state 0 from both states 1 and 3. The test designer must be aware of this, as a possible source of problems in the execution of tests.

V. APPLICATIONS OF THE SUPERTRACE AND TESTGEN PROGRAMS

In this chapter Simple Mushroom with Supertrace and Big Mushroom with Supertrace are demonstrated with several examples. Both programs are run with different protocols to give a specific view of the Supertrace algorithm.

In the first section, Simple Mushroom with Supertrace will be used to analyze a simple example four machine protocol which illustrates some basic aspects such as detecting unspecified receptions, unexecuted transitions etc. Then information transfer phase of a full duplex LAP-B protocol specified by the CFSM model will be analyzed. Later, the Big Mushroom with Supertrace will be used to analyze the Go Back N protocol with different window sizes and the Token Bus protocol, which illustrates important aspects of Supertrace algorithm.

In the second part of this chapter, an application of the protocol test sequence generator program (TESTGEN) to the well known FDDI protocol is illustrated.

A. Applications Of Mushroom Program With Supertrace

1. CFSM Model with Supertrace

a. Simple Four Machine Protocol

The specification of the protocol using the CFSM model is shown in Figure 40. This sample is chosen to demonstrate the coverage of supertrace algorithm with protocols that has relatively small number of states. Each machine sends/receives a message/acknowledgment from other machine. Machines 2 and 3 also have another send transition from state 1 to state 3. The FSM description of the protocol is shown in Figure 41 and analysis results obtained by the simple Mushroom with supertrace is shown in Figure 42. The analysis generated 36 global states. There are three unspecified receptions and one unexecuted transition. No deadlocks or channel overflows are recorded. The maximum channel size 2. These results are obtained by simply entering the FSM text file as an input to the program. This analysis would be difficult to do manually, even for a simple specification like this one.

The analysis results obtained is the same with simple mushroom [BULB93] results, showing the coverage and reliability of Supertrace for small protocols is around 100%.

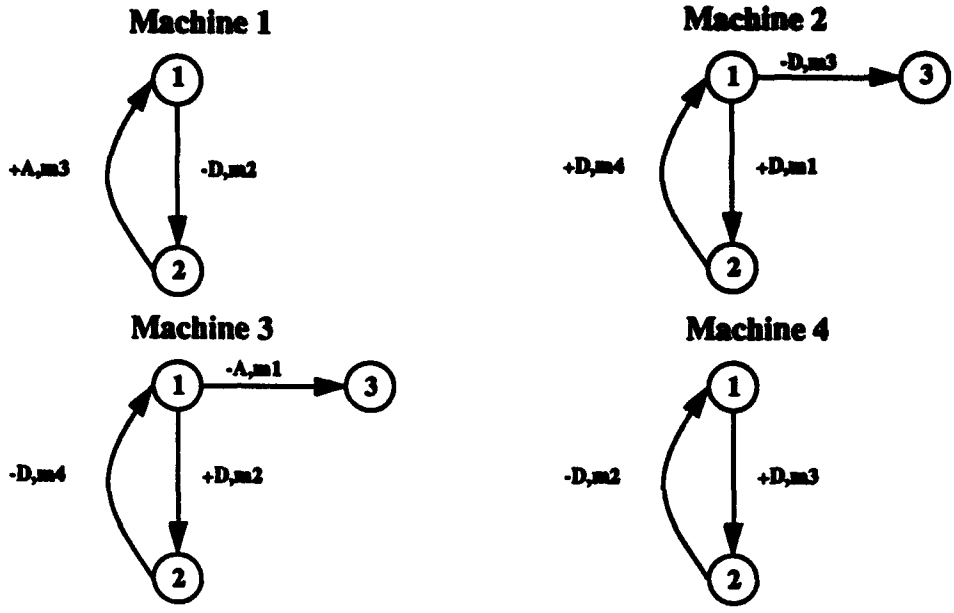

Figure 40 : Specification of the example four machine protocol

```
start
number_of_machines 4
machine 1
state 1
trans -D 2 2
state 2
trans +A 1 3
machine 2
state 1
trans -D 3 3
trans +D 2 1
state 2
trans +D 1 4
machine 3
state 1
trans -A 3 1
trans +D 2 2
state 2
trans -D 1 4
machine 4
state 1
trans +D 2 3
state 2
trans -D 1 2
initial_state 1 1 1 1
finish
```

Figure 41 : FSM text file for the example protocol

REACHABILITY ANALYSIS of : four_machine.fsm

SPECIFICATION

Machine 1 State Transitions			
From	To	other machine	Transition
1	2	2	s D
2	1	3	r A

Machine 2 State Transitions			
From	To	other machine	Transition
1	3	3	s D
1	2	1	r D
2	1	4	r D

Machine 3 State Transitions			
From	To	other machine	Transition
1	3	1	s A
1	2	2	r D
2	1	4	s D

Machine 4 State Transitions			
From	To	other machine	Transition
1	2	3	r D
2	1	2	s D

REACHABILITY GRAPH

```
1 [ 1,E,E,E, 1,E,E,E, 1,E,E,E, 1,E,E,E]
     -D   2  [ 2,D  ,E,E, 1,E,E,E, 1,E,E,E, 1,E,E,E]   2
     -D   3  [ 1,E,E,E, 3,E,D  ,E, 1,E,E,E, 1,E,E,E]   3
     -A   1  [ 1,E,E,E, 1,E,E,E, 3,A  ,E,E, 1,E,E,E]   4
2 [ 2,D  ,E,E, 1,E,E,E, 1,E,E,E, 1,E,E,E]
     -D   3  [ 2,D  ,E,E, 3,E,D  ,E, 1,E,E,E, 1,E,E,E]   5
     +D   1  [ 2,E,E,E, 2,E,E,E, 1,E,E,E, 1,E,E,E]   6
     -A   1  [ 2,D  ,E,E, 1,E,E,E, 3,A  ,E,E, 1,E,E,E]   7
3 [ 1,E,E,E, 3,E,D  ,E, 1,E,E,E, 1,E,E,E]
     -D   2  [ 2,D  ,E,E, 3,E,D  ,E, 1,E,E,E, 1,E,E,E]   0
     -A   1  [ 1,E,E,E, 3,E,D  ,E, 3,A  ,E,E, 1,E,E,E]   8
     +D   2  [ 1,E,E,E, 3,E,E,E, 2,E,E,E, 1,E,E,E]   9
4 [ 1,E,E,E, 1,E,E,E, 3,A  ,E,E, 1,E,E,E]
     -D   2  [ 2,D  ,E,E, 1,E,E,E, 3,A  ,E,E, 1,E,E,E]   0
     -D   3  [ 1,E,E,E, 3,E,D  ,E, 3,A  ,E,E, 1,E,E,E]   0
5 [ 2,D  ,E,E, 3,E,D  ,E, 1,E,E,E, 1,E,E,E]
     -A   1  [ 2,D  ,E,E, 3,E,D  ,E, 3,A  ,E,E, 1,E,E,E]  10
     +D   2  [ 2,D  ,E,E, 3,E,E,E, 2,E,E,E, 1,E,E,E]  11
6 [ 2,E,E,E, 2,E,E,E, 1,E,E,E, 1,E,E,E]
     -A   1  [ 2,E,E,E, 2,E,E,E, 3,A  ,E,E, 1,E,E,E]  12
7 [ 2,D  ,E,E, 1,E,E,E, 3,A  ,E,E, 1,E,E,E]
     +A   3  [ 1,D  ,E,E, 1,E,E,E, 3,E,E,E, 1,E,E,E]  13
     -D   3  [ 2,D  ,E,E, 3,E,D  ,E, 3,A  ,E,E, 1,E,E,E]   0
     +D   1  [ 2,E,E,E, 2,E,E,E, 3,A  ,E,E, 1,E,E,E]   0
8 [ 1,E,E,E, 3,E,D  ,E, 3,A  ,E,E, 1,E,E,E]
     -D   2  [ 2,D  ,E,E, 3,E,D  ,E, 3,A  ,E,E, 1,E,E,E]   0
9 [ 1,E,E,E, 3,E,E,E, 2,E,E,E, 1,E,E,E]
     -D   2  [ 2,D  ,E,E, 3,E,E,E, 2,E,E,E, 1,E,E,E]   0
     -D   4  [ 1,E,E,E, 3,E,E,E, 1,E,E,D  , 1,E,E,E]  14
10 [ 2,D  ,E,E, 3,E,D  ,E, 3,A  ,E,E, 1,E,E,E]
     +A   3  [ 1,D  ,E,E, 3,E,D  ,E, 3,E,E,E, 1,E,E,E]  15
11 [ 2,D  ,E,E, 3,E,E,E, 2,E,E,E, 1,E,E,E]
     -D   4  [ 2,D  ,E,E, 3,E,E,E, 1,E,E,D  , 1,E,E,E]  16
12 [ 2,E,E,E, 2,E,E,E, 3,A  ,E,E, 1,E,E,E]
     +A   3  [ 1,E,E,E, 2,E,E,E, 3,E,E,E, 1,E,E,E]  17
13 [ 1,D  ,E,E, 1,E,E,E, 3,E,E,E, 1,E,E,E]
     -D   2  [ 2,D  D  ,E,E, 1,E,E,E, 3,E,E,E, 1,E,E,E]  18
     -D   3  [ 1,D  ,E,E, 3,E,D  ,E, 3,E,E,E, 1,E,E,E]   0
     +D   1  [ 1,E,E,E, 2,E,E,E, 3,E,E,E, 1,E,E,E]   0
14 [ 1,E,E,E, 3,E,E,E, 1,E,E,D  , 1,E,E,E]
     -D   2  [ 2,D  ,E,E, 3,E,E,E, 1,E,E,D  , 1,E,E,E]   0
     -A   1  [ 1,E,E,E, 3,E,E,E, 3,A  ,E,D  , 1,E,E,E]  19
```

```
      +D   3  [ 1,E,E,E, 3,E,E,E, 1,E,E,E, 2,E,E,E]  20
15 [ 1,D  ,E,E, 3,E,D  ,E, 3,E,E,E, 1,E,E,E]
      -D   2  [ 2,D  D  ,E,E, 3,E,D  ,E, 3,E,E,E, 1,E,E,E]  21
16 [ 2,D  ,E,E, 3,E,E,E, 1,E,E,D  , 1,E,E,E]
      -A   1  [ 2,D  ,E,E, 3,E,E,E, 3,A  ,E,D  , 1,E,E,E]  22
      +D   3  [ 2,D  ,E,E, 3,E,E,E, 1,E,E,E, 2,E,E,E]  23
17 [ 1,E,E,E, 2,E,E,E, 3,E,E,E, 1,E,E,E]
      -D   2  [ 2,D  ,E,E, 2,E,E,E, 3,E,E,E, 1,E,E,E]  24
18 [ 2,D  D  ,E,E, 1,E,E,E, 3,E,E,E, 1,E,E,E]
      -D   3  [ 2,D  D  ,E,E, 3,E,D  ,E, 3,E,E,E, 1,E,E,E]   0
      +D   1  [ 2,D  ,E,E, 2,E,E,E, 3,E,E,E, 1,E,E,E]   0
19 [ 1,E,E,E, 3,E,E,E, 3,A  ,E,D  , 1,E,E,E]
      -D   2  [ 2,D  ,E,E, 3,E,E,E, 3,A  ,E,D  , 1,E,E,E]   0
      +D   3  [ 1,E,E,E, 3,E,E,E, 3,A  ,E,E, 2,E,E,E]  25
20 [ 1,E,E,E, 3,E,E,E, 1,E,E,E, 2,E,E,E]
      -D   2  [ 2,D  ,E,E, 3,E,E,E, 1,E,E,E, 2,E,E,E]   0
      -A   1  [ 1,E,E,E, 3,E,E,E, 3,A  ,E,E, 2,E,E,E]   0
      -D   2  [ 1,E,E,E, 3,E,E,E, 1,E,E,E, 1,E,D  ,E]  26
21 [ 2,D  D  ,E,E, 3,E,D  ,E, 3,E,E,E, 1,E,E,E]**********Unspecified Reception**********
22 [ 2,D  ,E,E, 3,E,E,E, 3,A  ,E,D  , 1,E,E,E]
      +A   3  [ 1,D  ,E,E, 3,E,E,E, 3,E,E,D  , 1,E,E,E]  27
      +D   3  [ 2,D  ,E,E, 3,E,E,E, 3,A  ,E,E, 2,E,E,E]  28
23 [ 2,D  ,E,E, 3,E,E,E, 1,E,E,E, 2,E,E,E]
      -A   1  [ 2,D  ,E,E, 3,E,E,E, 3,A  ,E,E, 2,E,E,E]   0
      -D   2  [ 2,D  ,E,E, 3,E,E,E, 1,E,E,E, 1,E,D  ,E]  29
24 [ 2,D  ,E,E, 2,E,E,E, 3,E,E,E, 1,E,E,E]**********Unspecified Reception**********
25 [ 1,E,E,E, 3,E,E,E, 3,A  ,E,E, 2,E,E,E]
      -D   2  [ 2,D  ,E,E, 3,E,E,E, 3,A  ,E,E, 2,E,E,E]   0
      -D   2  [ 1,E,E,E, 3,E,E,E, 3,A  ,E,E, 1,E,D  ,E]  30
26 [ 1,E,E,E, 3,E,E,E, 1,E,E,E, 1,E,D  ,E]
      -D   2  [ 2,D  ,E,E, 3,E,E,E, 1,E,E,E, 1,E,D  ,E]   0
      -A   1  [ 1,E,E,E, 3,E,E,E, 3,A  ,E,E, 1,E,D  ,E]   0
27 [ 1,D  ,E,E, 3,E,E,E, 3,E,E,D  , 1,E,E,E]
      -D   2  [ 2,D  D  ,E,E, 3,E,E,E, 3,E,E,D  , 1,E,E,E]  31
      +D   3  [ 1,D  ,E,E, 3,E,E,E, 3,E,E,E, 2,E,E,E]  32
28 [ 2,D  ,E,E, 3,E,E,E, 3,A  ,E,E, 2,E,E,E]
      +A   3  [ 1,D  ,E,E, 3,E,E,E, 3,E,E,E, 2,E,E,E]   0
      -D   2  [ 2,D  ,E,E, 3,E,E,E, 3,A  ,E,E, 1,E,D  ,E]  33
29 [ 2,D  ,E,E, 3,E,E,E, 1,E,E,E, 1,E,D  ,E]
      -A   1  [ 2,D  ,E,E, 3,E,E,E, 3,A  ,E,E, 1,E,D  ,E]   0
30 [ 1,E,E,E, 3,E,E,E, 3,A  ,E,E, 1,E,D  ,E]
      -D   2  [ 2,D  ,E,E, 3,E,E,E, 3,A  ,E,E, 1,E,D  ,E]   0
31 [ 2,D  D  ,E,E, 3,E,E,E, 3,E,E,D  , 1,E,E,E]
      +D   3  [ 2,D  D  ,E,E, 3,E,E,E, 3,E,E,E, 2,E,E,E]  34
32 [ 1,D  ,E,E, 3,E,E,E, 3,E,E,E, 2,E,E,E]
      -D   2  [ 2,D  D  ,E,E, 3,E,E,E, 3,E,E,E, 2,E,E,E]   0
      -D   2  [ 1,D  ,E,E, 3,E,E,E, 3,E,E,E, 1,E,D  ,E]  35
33 [ 2,D  ,E,E, 3,E,E,E, 3,A  ,E,E, 1,E,D  ,E]
      +A   3  [ 1,D  ,E,E, 3,E,E,E, 3,E,E,E, 1,E,D  ,E]   0
34 [ 2,D  D  ,E,E, 3,E,E,E, 3,E,E,E, 2,E,E,E]
      -D   2  [ 2,D  D  ,E,E, 3,E,E,E, 3,E,E,E, 1,E,D  ,E]  36
35 [ 1,D  ,E,E, 3,E,E,E, 3,E,E,E, 1,E,D  ,E]
      -D   2  [ 2,D  D  ,E,E, 3,E,E,E, 3,E,E,E, 1,E,D  ,E]   0
36 [ 2,D  D  ,E,E, 3,E,E,E, 3,E,E,E, 1,E,D  ,E]**********Unspecified Reception**********

SUMMARY OF REACHABILITY ANALYSIS (ANALYSIS COMPLETED)
-----------------------------------------------------
Total number of states generated : 36
Number of states analyzed : 36
number of deadlocks : 0
number of unspecified receptions : 3
maximum message queue size :  2
channel overflow :NONE
```

UNEXECUTED TRANSITIONS

Machine 2 Unexecuted Transitions			
From	To	other machine	Unexecuted Transition
2	1	4	r D

Figure 42 : Program Output for the Example

b. Analysis Of Information Transfer Phase Of The Lap-B Protocol

In this Section, analysis of a Data Link Control (DLC) protocol is described using the Simple Mushroom with Supertrace program. The physical layer of DLC (LAP-B) protocol was modeled and analyzed with CFSM model [LUND86].

The analysis of known protocols is important because it help us to determine the correctness and the coverage of the Supertrace algorithm. It is also an excellent example of how the total number of global states can grow very large, even for such a limited protocol.

This analysis demonstrates the main feature of the Supertrace algorithm, improved coverage, where there is insufficient memory available to conduct a full state analysis. The description of the information transfer phase is explained below as it appears in [LUND86].

The network nodes, which communicates by the protocol, consist of Data Terminal Equipment (DTE) and a Data Circuit Terminating Equipment (DCE). In this model, DTE and DCE are considered process 1 and process 2 respectively. Each of these processes are also modeled as three sub-processes: Sender, Receiver and Frame Assembler Disassembler (FAD).

Figure 43 shows the processes and their interrelationship. The FAD process combines data blocks, from the sender with acknowledgments from the Receiver, into complete I-frames. It sends the I-frames to the FAD of the other process. The FAD also parses received I-frames from the other FAD and sends the acknowledgment to the Sender, and data blocks to the Receiver.

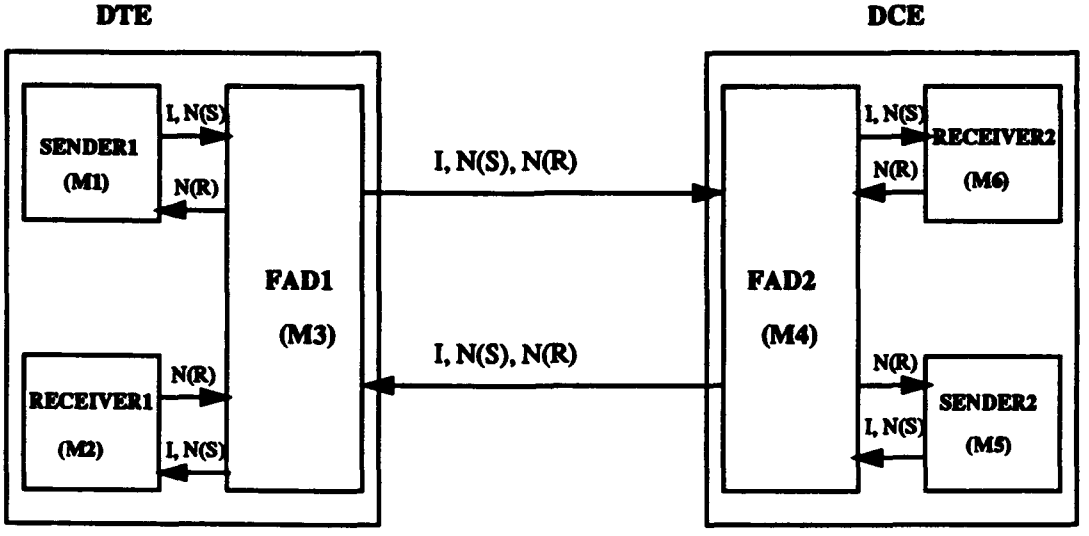

Figure 43 : Processes for the Information Transfer Phase

- **Model 1: I - frames only**

I-frames are expressed in the form "I_{nm}", where n is the send sequence number N(S), and m is the receive sequence number N(R). The message "Di" is a data block sent from the Sender to the FAD, or from the FAD to the receiver. It is this data block which is placed in or taken from, the I-frame. The 'i' in "Di" is the send sequence number. The message "Ai" is an acknowledgment with a receive sequence number of 'i'. The finite state machines for the Sender, Receiver and FAD of the DTE are shown in Figures 44, 45, and 46. The FSMs for the DCE are the same with a 2 substituted for 1 wherever it occurs. Since no RR-frames are used, I-frames can only be acknowledged by receiving an N(R) from an incoming data frame.

- **Model 2: I - frames and RR's**

If the DCE does not have any user data blocks to send, it is not able to acknowledge the receipt of the DTE I-frames. In this case, the DTE should stop sending frames after it reaches the window limit.

The solution to this problem is the Receive Ready, or "RR" message. It is an S-frame, containing no user data block, but does contain an acknowledging sequence number. Its purpose is to inform the receiving process (DTE in this case) that the sending process (DCE) is ready to receive the I-frame numbered N(R); it acknowledges I-frames up to N(R) - 1. The Receiver1 with I and RR frames is shown in Figure 46. The FAD with RR frames are specified by dashed transitions in Figure 47.

In the Receiver1 there are now two types of acknowledgment messages: "ACKi," and "Ai," for i = 0, 1, 2; in the first model we had only "Ai". This is to allow for two different ways of acknowledging I-frames by the Receiever1 process: by I-frames or by RR-frames.

When the FAD process has data to send, it queries the Receiver by sending an "ENQ"; this insures that the latest N(R) is sent along with the I-frame. These enquiries are answered by an "Ai" message. But if the FAD process has no data to send, it has no way of knowing whether any I-frames have been received and need to be acknowledged. This is the purpose of the "ACKi" messages; to allow the Receiver to initiate an acknowledgment.

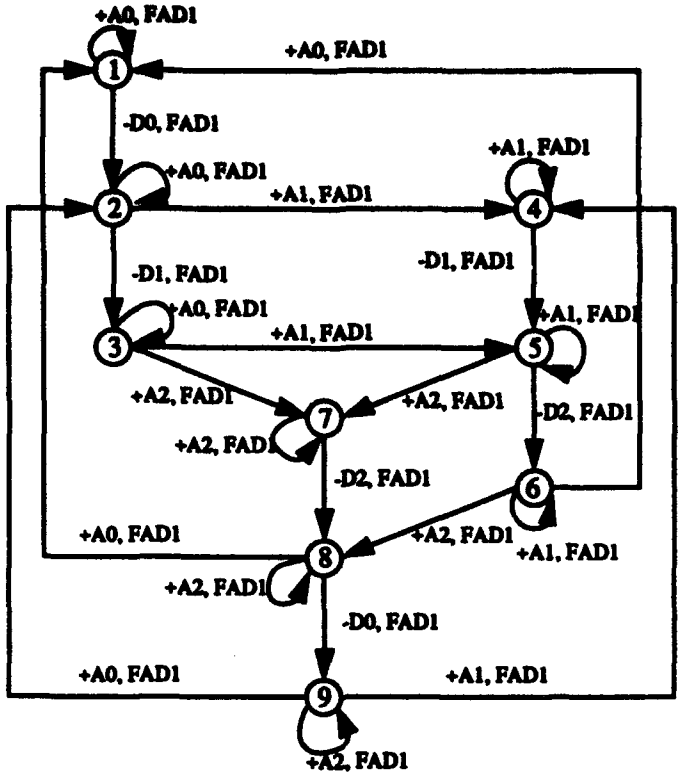

Figure 44 : Sender 1 of LAP-B Protocol

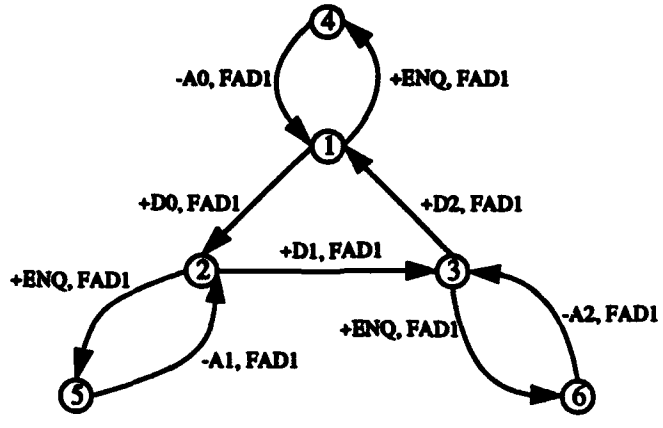

Figure 45 : Receiver 1 of LAP-B Protocol (I-frames only)

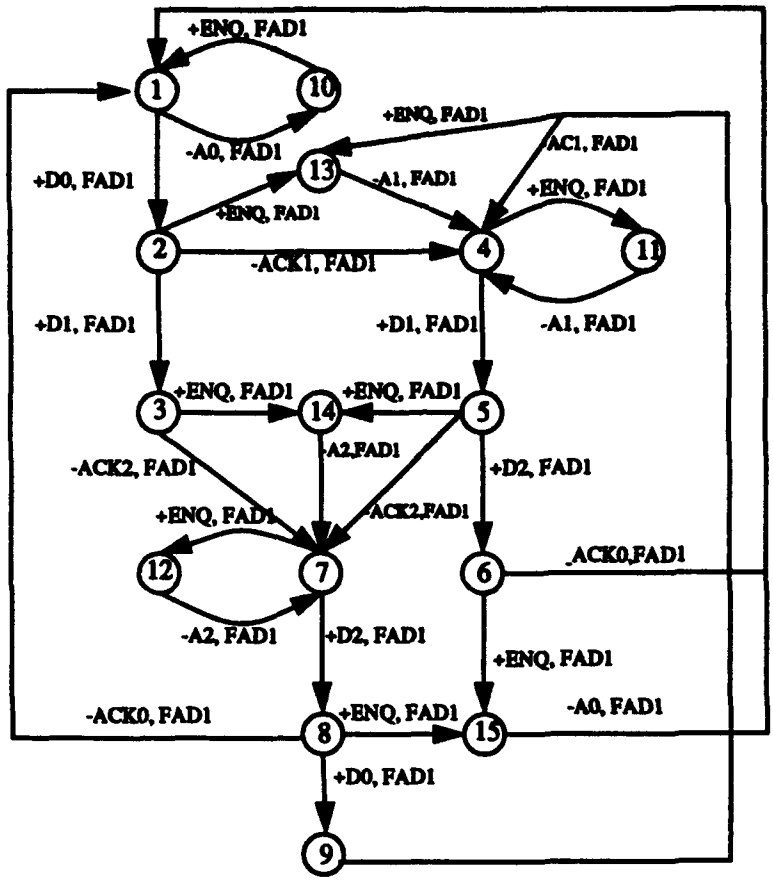

Figure 46 : Receiver 1 of LAP-B Protocol (I and RR Frames)

For the automated analysis, the FSMs in Figures 44,45,46 and 47 are converted to a text file and entered into program. The transition names in this text file are the same as in the FSM diagrams except, transition arc "ACKi" is represented as "ACi."

The program was run with two different input files the LAP-B protocol with I-frames and Lap-B protocol with I and RR frames. At the end of analysis 69102 states from the Lap-B protocol with I-frames were generated and analyzed. No unspecified receptions, unexecuted transitions or channel overflows were discovered. The maximum channel length was 6.

A deadlock condition was found at state 16817. All channels were empty and Sender1, Receiver1, FAD1, FAD2, Sender2, Receiver2 were in states 3, 3, 1, 1, 3, 3 respectively. The state deadlock was expected since RR-frames were not included in this analysis. The main difference between the analysis results with supertrace and the full state analysis of the protocol [BULB93], is the number of states generated and analyzed. The number of states generated with full state search algorithm was 73391. The supertrace algorithm generated almost 95% (69102/

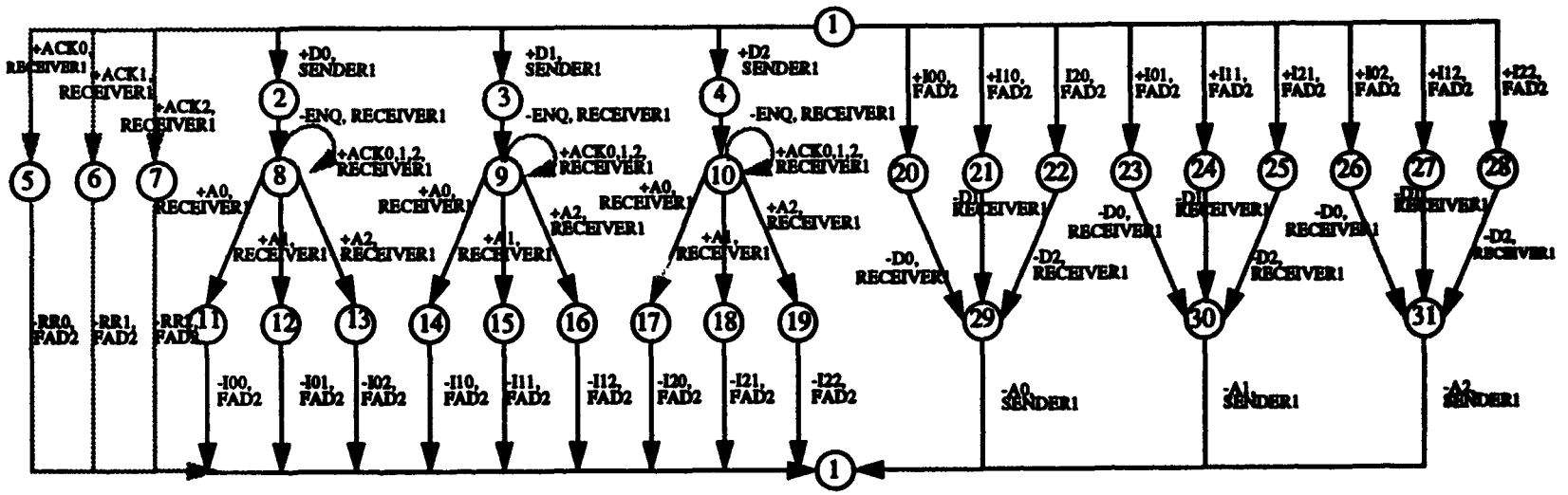

Figure 47 : FAD 1 of The LAP-B Protocol

73391 = 0.941) of all the states. The size of the memory is a critical factor in the generation of collisions. The algorithm provides better coverage with a larger hash table and effective hash function.

In the analysis of the same protocol utilizing the regular mushroom program, the deadlock was detected at state number 17034. The difference of 217 states between the two programs, does not necessarily mean that 217 collisions occurred. It is possible, though not probable, that one collusion occurred and 216 successor states were never considered. We do know that the number of collusions is between 1 and 217. It should be emphasized that the purpose of the supertrace program is not to produce a total coverage of states. The purpose is to validate those network protocols through a controlled partial search which cannot be exhaustively analyzed.

The LAP-B Protocol, including RR-frames, was also analyzed. The program could not complete the analysis due to insufficient memory. At the point of termination 300456 global states had been generated and analyzed. No unspecified receptions, deadlocks or channel overflows were recorded for the analyzed portion of the protocol. The maximum channel size reached was 5. The number of states generated with regular mushroom program on the same protocol was 153565 [BULB93]. These results clearly show the improvement of the supertrace algorithm option over the regular mushroom. 146891 more states are generated and analyzed by Supertrace algorithm.The 96% increase in the number of states analyzed, is a clear indication of the improvement of the Supertrace algorithm over regular Mushroom program. A sample input for LAP-B protocol with I and RR frames and partial analysis results are shown in Appendix A.

2. SCM Model With Supertrace

There are a few programs specified formally by SCM model which have been analyzed by Big mushroom program in [BULB93]. The same specifications will be used to make a comparison of regular and big mushroom with supertrace.

a. Go Back N Protocol

The protocol selected for analysis is a one way data transfer protocol with a variable window size, which is essentially a subset of the High-Level Data Link Control(HDLC) class of protocols. This model is modeled and analyzed in [LUND91][BULB93]. The same specification with different window sizes was used to compare the supertrace and exhaustive search algorithms.

The summary of the specification is explained below. There are two machines in the system, a sender (m_1) and a receiver (m_2). The sender sends data blocks to the receiver, which are numbered sequentially, 0, 1,....., w, 0, 1, ... for a window size of w. As in HDLC, the maximum

number of data blocks which can be sent without receiving an acknowledgment is w, the window size. The receiver, m2, receives the data blocks and acknowledges them by sending the sequence number of the next data block expected (which is stored in local variable expel). The shared variables *DATA* and *SEQ* are used to pass messages from sender to receiver, and the shared variable *ACK* is used to pass acknowledgments back to the sender. The receiver may acknowledge any number of blocks received up to the window size. Upon receiving the acknowledgment, the sender must be able to deduce how many data blocks are being acknowledged. This is done by observing the difference between the values of the received acknowledgment and the sequence number of the last data blocks sent.

The general specification of the protocol is given in Figure 48 and in Table 6. Initially, both sender and receiver are in state 0, arrays *DATA* and *SEQ* are empty, and *ACK* is empty. The domains of *DATA*, *Rdata* and *Sdata* are not specified; these are used to hold user data blocks. *Sdata* and *Rdata* are the interface or access points of the higher layer protocol. The local variables for the sender are *Sdata*, used to store data blocks, *seq*, used to store the sequence number of the next data block to be sent out, and i, used as an index into the *DATA* and *SEQ* arrays. Initially *seq* is set to 0, and i is set to 1. The local variables of the receiver are *Rdata*, *exp*, and *j*. *Rdata* is used to receive and store incoming data blocks, exp to hold the expected sequence number of the next incoming data block, and *j* is an index into the shared arrays *DATA* and *SEQ*.

There are four basic types of transitions. In the sender, m1, the -D transition transmits a data block by placing it into the shared variable *DATA(i)*, and the sequence number into *SEQ(i)*. The send is enabled whenever those variables are empty.(The interaction between the sender and the user, or higher layer is not specified here). The *inc* operation increments its arguments, if less than their maximum value, in which case it resets them to the minimum value. The operator "" $\oplus$ "" represents the *inc* operation repeated k times, if the argument is k and the symbol ε denotes the empty value. The receive transition in the receiver, m2, is enabled whenever a data block of the appropriate sequence number is in the *jth* element of *DATA* and *SEQ*. An acknowledgment may be sent by m2 in any state except 0, in which case no acknowledged data blocks have been received.

The remaining transition is the +Ak receive acknowledgment, in m1. If m1 is in state u, $1 \leq u \leq w$, and there is nonempty value in shared variable *ACK*, then exactly one of the transitions +A0, +A1, ..., +Aw-1 will be enabled; it will be that Ak such that the predicate $\mathrm{ACK} \oplus k = seq$ is true, and the next state is k[LUND91].

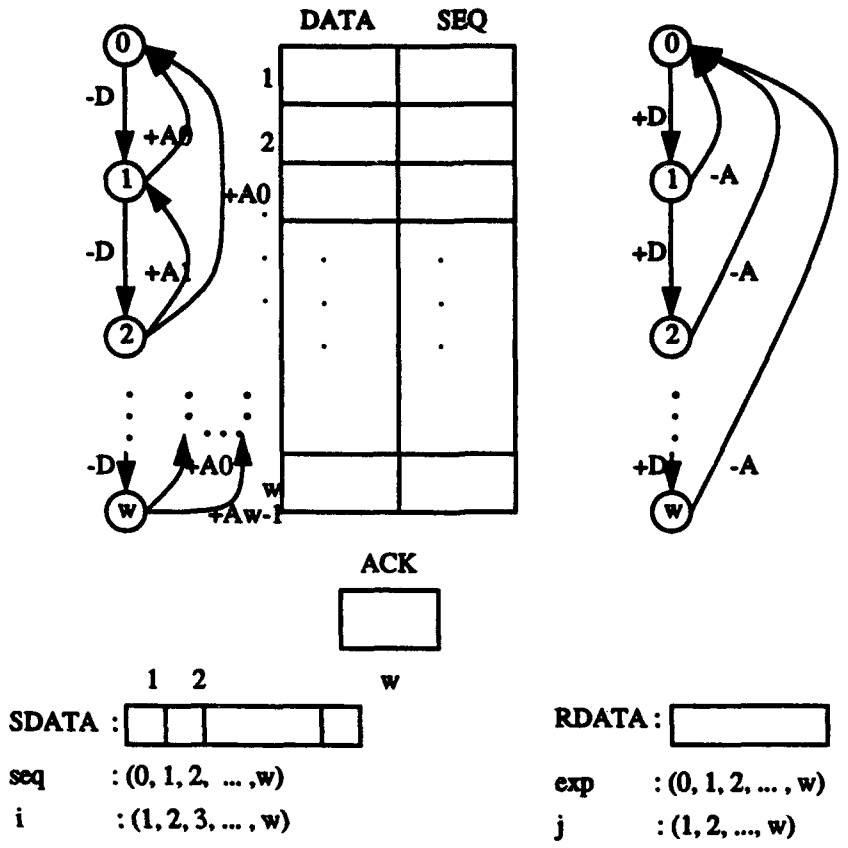

Figure 48 : State Machine and Variables of the Go-Back-N Protocol

TABLE 6: PREDICATE ACTION TABLE OF GO-BACK-N PROTOCOL

Transition	Enabling Predicate	Action
-D	DATA(i) = ε ∧ SEQ(i) = ε	DATA(i) := Sdata(i) SEQ(i) := seq inc(i,seq)
$+A_k$ $(0 \le k \le w)$	ACK ⊕ k = seq ∧ ACK ≠ ε (next state :k)	ACK := ε
+D	DATA(j) ≠ ε ∧ SEQ(j) = exp	Rdata := DATA(j) DATA(j), SEQ(j) := ε inc(j,exp)
-A	DATA(j) = ε	ACK := exp Rdata := ε

For analyzing this protocol by Big Mushroom with Supertrace program, the inputs to the program should be completed. These consist of a text file description of FSMs, the package, definitions, which include the variables of the protocol, and the subprograms

Analyze_Predicate_Machines and *Action*, which define the *predicate-action table* and *Output_Gtuple procedure*, which defines the output format for the global tuples, must be entered. The user should also write the *Global_hash* function in Ada Programing language that covers local and shared variables and machine states of the protocol. Completed packages/procedures and global hash function for a window size of 10 are given in Appendix B.

The same names are used for local and shared variables in the package definitions as in the *predicate-action table*. Variables *DATA*, *ACK* and *Sdata* are declared as one dimensional array of window size. Local variables *seq* and *exp* and index numbers *i* and *j* are declared as integers in the range 0 to window size. Global variable *ACK* is declared as integer in the range -1 to window size, where -1 represents ε value in the predicate action table. An enumeration type, *buffer_type*, is declared for storing the data passed by the upper layer to local variable *Sdata*. Data are declared as d0, d1, ..., d9,e, where e represents the ε value. Transition names in the specification are defined as *send_data*, *rcv_data*, *snd_ack*, *Rcv_acki* for *-D*, *+D*, *-A*, and *+Ai* in *predicate-action table* respectively.

The global state analysis of Go-Back-N protocol with different window sizes was conducted by both Big Mushroom and Supertrace algorithms. The number of global states generated in these programs is listed in Table 7 ("WS" represents the window size). In the analysis of the Go-Back-N protocol with a window size of 18, Big Mushroom program was interrupted due to a memory error and could not complete the analysis. No deadlocks, unexecuted transitions or Channel overflows were encountered in the analyzed portion of the protocol. The comparison of these results and the advantages of Supertrace algorithm will be discussed in Chapter V.

TABLE 7: THE NUMBER OF STATES GENERATED WITH BIG MUSHROOM AND SUPERTRACE ALGORITHM

GBN Protocol	WS =10	WS=12	WS=13	WS=14	WS=18
Big Mushroom	31460	70980	101920	142800	161431
Supertrace	30632	66654	90210	122880	290980
Coverage of Super-trace	97%	94%	89%	86%	Unknown

b. Token Bus Protocol

Another example of the program application, the token bus specification in [CHAR90] will be used. The specification is a simplified one, which will be used to demonstrate

the coverage of Supertrace algorithm for protocols with small number of states. It assumes that the transmission medium is error free and all transmitted messages are received undamaged. The global state analysis is generated from this token bus specification for a protocol consisting of 8 machines.

The specification of the protocol is given in Figure 49 and Table 8. The FSM diagram and the local variables are the same for each machine, where the transition names: *ready*, *rcv*, *pass*, *get-tk*, *pass-tk*, *Xmit*, and *moreD* are appended with the corresponding machine number to the end of each machine in the specification. This makes it easier to follow the reachability graphs. The remainder of the protocol specification as described in [CHAR90] is as follows: The shared variable, *MEDIUM*, is used to model the bus, which is "shared" by each machine. A transmission onto the bus is modeled by a write into the shared variable. The fields of this variable correspond to the parts of the transmitted message: the first field, *MEDIUM.T*, takes the values *T* or *D*, which indicate whether the frame is a token or a data frame. The second field contains the address of the station to which the message is transmitted (*DA* for "destination address"); the next field, the originator (*SA* for "source address"); and finally the data block itself.

The network stations, or machines, are defined by a finite state machine, a set of local variables, and a predicate-action table. The initial state of each machine is state 0, and the shared variable is initially set to contain the token with the address of one of the stations in the "*DA*" field.

The value of local variable next is the address of the next or downstream neighbor, these are initialized so the entire network forms a cycle, or logical ring.

The local variable *i* is used to store the station's own address. As implied by the names, the local variables *inbuf* and *outbuf* are used for storing data blocks to be transmitted to or retrieved from other machines on the network. The latter of these, *outbuf*, is an array and thus can store a potentially large number of data blocks. The local variable *ctr* serves to count the number of blocks sent; it is an upper bound on the number of blocks which can be sent during a single token holding period. The local variable *j* is an index into the array *outbuf*.

The local variables *j* and *ctr* are initially set to 1, and *inbuf* and *outbuf* are initially set to empty. The shared variable *MEDIUM* initially contains the token, with the address of the station in the *DA* field. Thus the initial system state tuple is (0, 0, ..., 0) and the first transition taken will be *get_tk* by the station which has its local variable i equal to *MEDIUM.DA*.

Each machine has four states. In the initial state, 0, the stations are waiting to either receive a message from another station, or the token. If the token appears in the variable MEDIUM with the station's own address, the transition to state 2 is taken. When taking the *get-tk* transition, the machine clears the communication medium and sets the message counter *ctr* to 1. In state 2, the station transmits any data blocks it has moving to state 3, or passes the token, returning to state 0.

In state 3, the station will return to state2 if any additional blocks are to be sent, until the maximum count k is reached, or when all the stations' messages have been sent, the station returns to state 0.

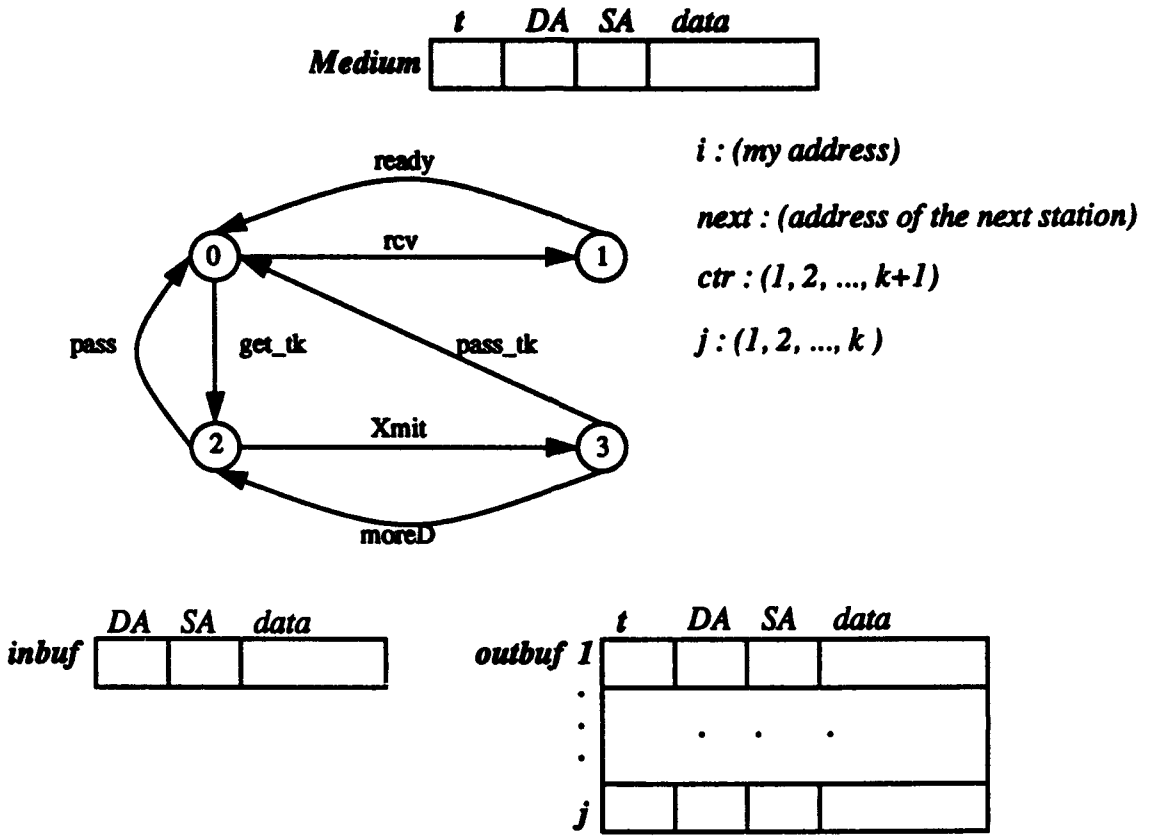

Figure 49 : FSM and Variables of Token Bus Protocol

The receiving station, as with all stations not in possession of the token, will be in state 0. The message will appear in MEDIUM, with the receiving station's address in the DA field. The receiving transition to state 1 will then be taken, the data block copied, and MEDIUM cleared. By clearing the medium, the receiving station enables the sending station to return to its initial state (0) or to its sending state (2).

TABLE 8: PREDICATE ACTION TABLE FOR TOKEN BUS PROTOCOL

Transition	Enabling Predicate	Action
rcv	MEDIUM.(t,DA)= (D, i)	inbuf := MEDIUM.(SA,data)
ready	true	MEDIUM := Ø
get-tk	MEDIUM.(t,DA) = (T, i)	MEDIUM:= Ø; ctr := 1
pass	outbuf[j] = Ø	MEDIUM := (T, next, i, Ø)

TABLE 8: PREDICATE ACTION TABLE FOR TOKEN BUS PROTOCOL

Transition	Enabling Predicate	Action
Xmit	outbuf[j] ≠ Ø	MEDIUM := outbuf[j]; ctr := ctr ⊕ 1 ; j := j ⊕ 1; outbuf[j] := Ø
moreD	MEDIUM ∧ Ø = outbuf[j] ≠ Ø	null
pass-tk	MEDIUM = Ø ∧ "" (outbuf[j] = Ø ∨ ctr = k + 1)	MEDIUM := (T, next, i, Ø)

The symbol "Ø" indicates that the variable should be incremented unless its maximum value has been reached, in which case it should be reset to the initial value.The notation MEDIUM.(t, DA) is used to denote the first two fields of the variable MEDIUM. For example, MEDIUM.(t, DA) = (T, i) is a boolean expression which is true if and only if the first fields of MEDIUM contains the value T, and the second field contains the value i. Other notations in the predicate-action table are intuitive.

The same names as in the specification are used for the local and global variables in the package definitions. Also, the "empty" value is represented by "E" and the data are represented by "I" in this package. The upper bound on the number of the data blocks in the *outbuf* variable is set to 7.

The results are same with the previous analysis results [BULB93].The global state analysis with supertrace has generated 263 global states and there were no deadlocks or unexecuted transitions.

B. Automated Test Generation Of FDDI Protocol By "TESTGEN" Program

In this section an automated test generation of the FDDI protocol is illustrated. FDDI is a standard for a 100Mb/s fiber optic network which has come on the market in the last few years. The protocol was formally specified, including timing requirements, and verified, in [LUND90B]. The same specification of FDDI protocol will be used in this section. The brief description of the FDDI protocol is given below.

The protocol specification consists of the FSM description of each machine,Figure 50; the predicate-action table (Table 9); and the timer specifications (not shown). A detailed description of protocol appears in [LUND90B], so here we give only a brief description.

Each machine shares one variable with its upstream neighbor (called *inbuf*) and one with its downstream neighbor (called *outbuf*). (These shared variables serve as the input and output ring connections).

The FSM consists of 20 states. In states 0-7 the station has nothing to transmit, so is merely waiting for the token and processing it. In states 10-21 the station has a message to transmit, and does so upon receiving the token. The transition names on the transition arcs serve as a key into the PAT, which specifies the action taken when the transition is executed.

There are two transitions specified in the Table 9 which are not shown in the state diagram; this is because these transitions can be taken from any state.The *TRT-watch* transition becomes enabled whenever the *TRT* timer expires. This transition immediately resets the timer, and increments variable *Late-cnt*. The second transition not shown is called *CRASH*; this is the termination of the ring operation, which occurs if the token fails to circulate within twice the *TTRT*.

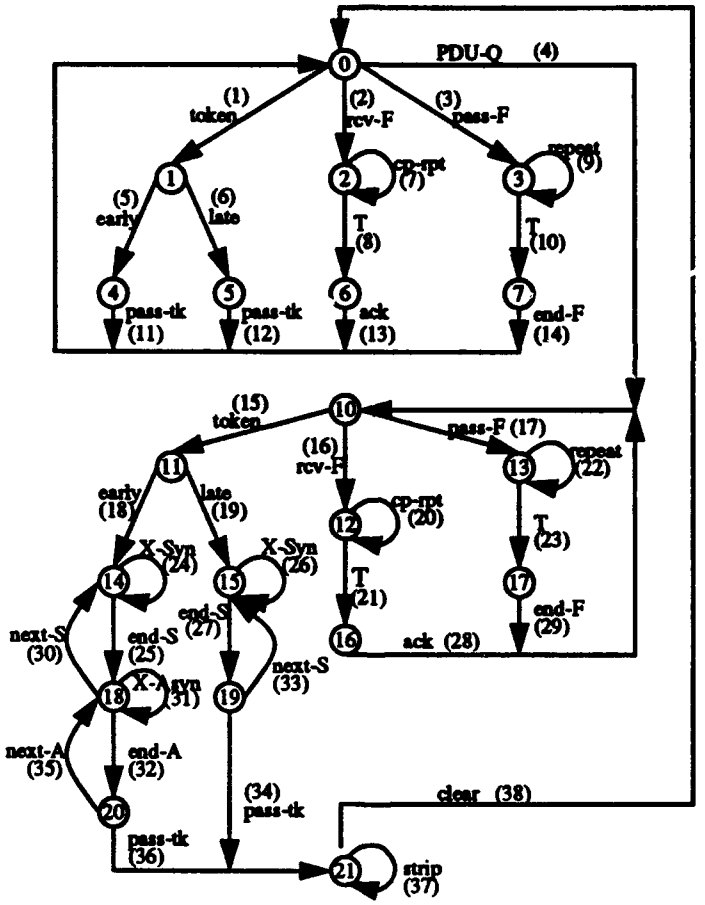

Figure 50 : FSM of the FDDI Protocol

TABLE 9: PREDICATE ACTION OF THE FDDI PROTOCOL

Transition	Enabling predicate	Action
PDU-Q	A-buf(i) ≠ Ø ∨ S-buf(j) ≠ Ø	
token	inbuf[1..7] = (I, J, K, 0, 0, T, T)	inbuf:=Ø; S-cnt:=0
early	Late-cnt = 0	THT-val := TRT-val; TRT-val := T-Opr
late	Late-cnt > 0	Late-cnt := 0;
pass-tk	TRUE	outbuf[1..7] := (I, J, K, 0, 0, T, T)
rcv-F	inbuf[5] ∈ {1, 2} ∧ inbuf[6..7] = MA	in := 1
cp-rpt	inbuf[in] ≠ T	msg-buf[in],outbuf[in]:= inbuf[in]; in :=in+1
T	inbuf[in] = T	outbuf[in] := T; inbuf := Ø; in := in +1
end-F	TRUE	outbuf[in,in+1,in+2]:= (err,inbuf[in+1,in+2])
ack	TRUE	outbuf[in, in+1, in+2] := (err, 1, 1)
pass-F	inbuf[5] ∈ {1, 2} ∧ inbuf[6..7] ≠ MA	in := 1
repeat	inbuf[in] ≠ T	outbuf[in]:= inbuf[in]; in := in+1
X-Syn	S-buf[j,out] ≠ Ø	outbuf[out] := S-buf[j,out]; out:= out+1
X-Asyn	A-buf[i,out] ≠ Ø ∧ (S-cnt = max ∨ S-buf[j] = 0)	outbuf[out] := A-buf[i,out]; out:= out+1
end-S	S-buf[j,out] = Ø	outbuf[out,out+1,out+2] := (T,0,0); S-cnt,F-cnt := S-cnt+1; j, out := j ⊕ 1, 1
end-A	A-buf[i,out] = Ø	outbuf[out,out+1,out+2] := (T,0,0); F-cnt:= F-cnt+1; i, out := i ⊕ 1, 1
next-S	S-cnt < max ∧ S-buf[j] ≠ Ø	
next-A	THT-val > 0 ∧ A-buf[i] ≠ Ø	
strip	inbuf[6..7] = MA ∧ F-cnt > 0	inbuf:= Ø; F-cnt := F-cnt - 1
clear	F-cnt = 0	
TRT-watch	TRT-val = 0	TRT-val := T-opr; Late-cnt := Late-cnt+1
CRASH	Late-cnt > 1	terminate ring operation

1. Creating the FSM And Predicate-Action Input Files for the FDDI Protocol

Creation of the FSM input file is a straightforward process. The user should number all transitions on the finite state machine as shown in Figure 50. All transitions should be written to a input text file according to the rules in Chapter IV. The FSM input file for the FDDI protocol is shown in Figure 51 and Predicate-action input file is shown in Figure 52.

Some of the relational symbols in the Predicate-Action Table are converted to their semantically equivalent text forms. For example relational symbols $\wedge$, $\vee$ are converted to "and" and "or" respectively. A relatively more complex symbol i:= i $\oplus$ 1 is represented as "i := i(mod+)1."

The TESTGEN program first prints out all the paths in the protocol. It also finds all the cycles and checks them for a transition that will ultimately lead back to the initial state. All possible paths in the FDDI protocol are output to a file as shown in Figure 53. The paths are depicted according to the numbers assigned by the user.

```
0
0 1 1 token
0 2 2 rcv-f
0 3 3 pass-f
0 10 4 pdu-q
1 4 5 early
1 5 6 late
2 2 7 cp-rpt
2 6 8 t
3 3 9 repeat
3 7 10 t
4 0 11 pass-tk
5 0 12 pass-tk
6 0 13 ack
7 0 14 end-f
10 11 15 token
10 12 16 rcv-f
10 13 17 pass-f
11 14 18 early
11 15 19  late
12 12 20 cp-rpt
12 16 21 t
13 13 22 repeat
13 17 23 t
14 14 24 x-syn
14 18 25 end-s
15 15 26 x-syn
15 19 27 end-s
16 10 28 ack
17 10 29 end-f
18 14 30 next-s
18 18 31 x-asyn
18 20 32 end-a
19 15 33 next-s
19 21 34 pass-tk
20 18 35 next-a
20 21 36 pass-tk
21 21 37 strip
21 0 38 clear
```

Figure 51 : FSM Input File of FDDI Protocol

```
pdu-q    | a-buf(i) /= 0 or s-buf(j) /= 0 | no                                                    |
token    | inbuf(1..7) = (i,j,k,0,0,t,t)  | inbuf := 0 ; s-cnt := 0                               |
early    | late-cnt = 0                   |tht-val := trt-val ; trt-val := t-opr                  |
late     | late-cnt > 0                   | late-cnt := 0                                         |
pass-tk  | true                           | outbuf(1..7) := (i,j,k,o,o,t,t)                       |
rcv-f    |  inbuf = (x,x,x,x,1or2,ma)     |  in := 1                                              |
cp-rpt   | inbuf(in) /= t | msg-buf(in)   := inbuf(in) ; outbuf(in) := inbuf(in) ; in := in+1 |
t        | inbuf(in) = t                  | outbuf(in) := t ; inbuf := 0 ; in := in+1 |
end-f    | true                           | outbuf(in,in+1,in+2] := (err,inbuf(in+1,in+2]) |
ack      | true                           | outbuf(in,in+1,in+2] := (err,1,1)     |
pass-f   |  inbuf = (x,x,x,x,1or2,/=ma) | in := 1                                               |
repeat   | inbuf(in] /= t                 | outbuf(in] := inbuf(in] ; in := in+1 |
x-syn    | s-buf[j,out] /= 0              | outbuf[out] := s-buf[j,out] ; out := out+1 |
x-asyn   | a-buf[i,out] /= 0 and ( s-cnt = max or s-buf[j] = 0 ) | outbuf[out] := a-buf[i,out];out:=out+1|
end-s    | s-buf[j,out] = 0 | outbuf[out,out+1,out+2] := (t,0,0) ; s-cnt := s-cnt+1 ; f-cnt := f-cnt+1  |
end-a    | a-buf[i,out] = 0 | outbuf[out,out+1,out+2] := (t,0,0) ; f-cnt := f-cnt+1 |
next-s   | s-cnt < max and s-buf[j] /= 0 | no                                                    |
next-a   | tht-val > 0 and a-buf[i] /= 0 | no                                                    |
strip    | inbuf(6..7) = ma and f-cnt > 0 | inbuf := 0 ; f-cnt := f-cnt+1                        |
clear    | f-cnt = 0                      | no                                                    |
trt-watch | trt-val = 0                   | trt-val := t-opr ; late-cnt := late-cnt+1 |
crash    | late-cnt > 1                   | no                                                    |
```

Figure 52 : Predicate Action Input File of FDDI Protocol

```
1  5 11  is another path
1  6 12  is another path
2  8 13  is another path
2  7  8 13  is another path
3 10 14  is another path
3  9 10 14  is another path
4 15 19 27 34 38  is another path
4 15 19 27 34 37 38  is another path
4 15 19 26 27 34 38  is another path
4 15 19 26 27 34 37 38  is another path
4 15 18 25 32 36 38  is another path
4 15 18 25 32 36 37 38  is another path
4 15 18 24 25 32 36 38  is another path
4 15 18 24 25 32 36 37 38  is another path
                  .
                  .
                  .
                  .
4 17 22 23 29 16 21 28 15 18 25 31 32 36 38  is another path
4 17 22 23 29 16 21 28 15 18 25 31 32 36 37 38  is another path
4 17 22 23 29 16 21 28 15 18 24 25 31 32 36 38  is another path
4 17 22 23 29 16 21 28 15 18 24 25 31 32 36 37 38  is another path
4 17 22 23 29 16 20 21 28 15 18 25 31 32 36 38  is another path
4 17 22 23 29 16 20 21 28 15 18 25 31 32 36 37 38  is another path
```

Figure 53 : The Representation of Paths in the Output File for the FDDI Protocol

In our example, the number of paths found by the TESTGEN program is 162. There are no cycles without an outgoing transition that leads back to the initial state.

Finally, the TESTGEN program creates the testing sequence table by printing all possible transition sequences, excluding continuous cycles. The table is 2112 lines long. Since the size of the table generated for the FDDI protocol is too big to show here, it is partially depicted in Figure 54.

Each of these 2112 output lines corresponds to a single test. In Figure 55 only the first few test are shown. The width of the table corresponds to the number of input and output variables.

Figure 54 : Partial Output of FDDI Protocol

For example, consider the first test in Figure 55. The start state, S_I, is state 0; the end state, S_E, is

state 1. The input variables must be set to the values shown on the left side of the table, and the output variables are expected to take on the values shown on the right side. The next test will take us to state 4.

If a variable is both input and output it is marked by (i) for input and (o) for output variable to show their status in the generated test sequence. For example, *late_count* appears both in the enabling predicate and in the action part of transition "*late*". It is both an input and output variable and is thus represented in the output test sequence as *late_cnt(i)* and *late_cnt(o)* as in Figure 54.

If there is more than one clause in the enabling predicate part of the predicate action table the TESTGEN program generates one test sequence and marks the variables of this test with the clause's relational symbol. In our example enabling predicate for the *PDU-Q* transition consists of two clauses. The TESTGEN program illustrates this by putting the relational symbol "*or*" (relational symbol in this case) in front of the values to be compared in the output file. The values for *a-buf (or /=empty) and s-buf (or/=empty)* should be read as "A-buf is not equal to empty or S-buf is not equal to empty." It is the responsibility to the user to change the variables for that transition to enable that transition. For testing purposes, the user can either make one or both of these two variables non-empty.

If there are more than two clauses in the enabling predicate part of the *PAT* as mentioned in Chapter IV, the TESTGEN program is able to represent these clauses in the output test sequence table. In the FDDI PAT (TABLE 9), the *X-Asyn* transition has more than two clauses in the format "first clause relational symbol (second clause relational symbol third clause)." The TESTGEN program shows this in the output sequence by putting the relational symbol in parentheses to represent the symbol between the second and third clauses, and placing the first relational symbol without parentheses in the output file. For example, the *a-buf[i,out]* has a value "*=/empty*," *s-buf* has a value "*(or)empty*" and *s-count* has a value "*(or) max*" in the generated test sequence. This test sequence input should be read as "*A-buf[i,out]* should not be empty and either *S-cnt* should be equal to *max* or *S-buf[j]* should be empty."

The TESTGEN program can determine some transitions which make a state transient. It informs the user by printing out a warning to the terminal and output file. In our example, the TESTGEN detects "*end-f*" and "*ack*" transitions, which makes states 6, 7, 16 and 17 transient, and prints out a warning.

Since the TESTGEN program generates all possible transition sequences, returning to the initial state, protocol testing can be executed by following the order of tests in the test sequence

file. This means that there is no need to find the *UIO (unique input-output)* sequence after each individual test, but only at the end of the last test (or possibly not at all).

Finally, the TESTGEN program also detects converging transitions, if any, and prints out the list of the converging transitions. In the case of FDDI protocol, *pass-tk* is detected as a converging state from states 4-5 and also from states 19-20. The test designer should be aware of this as a possible source of problems in the execution of tests.

VI. CONCLUSION AND FURTHER RESEARCH POSSIBILITIES

In this chapter both software tools' capabilities are summarized and further research possibilities are discussed.

A. Supertrace Algorithm

In the first part of this thesis a software tool has been described which improves the automatic analysis of protocols specified by the CFSM and SCM models, by using the Supertrace algorithm.

This algorithm improves the coverage of protocol analysis by generating a larger number of states than regular mushroom program. In cases where exhaustive search algorithm is infeasible, this can be extremely helpful. It also shows that the mushroom program with supertrace is capable of covering up to 95% for protocols with 1.5×10^5 global states. The improvement of the Supertrace algorithm is illustrated in Figure 55 and Figure 56. The protocols are represented in abbreviated form (i.e. Gbn for the Go-Back-N protocol).The number of states generated by mushroom with supertrace is between 90% and 95% for protocols up to 150000 global states and around 99% for protocols with 20000 global states.

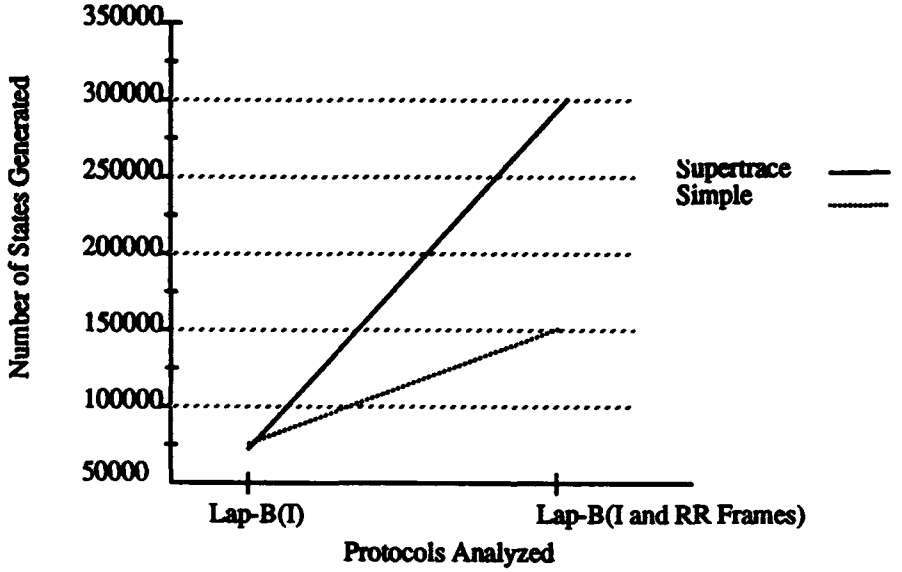

Figure 55 : The Analysis Results of Supertrace and Simple Mushroom

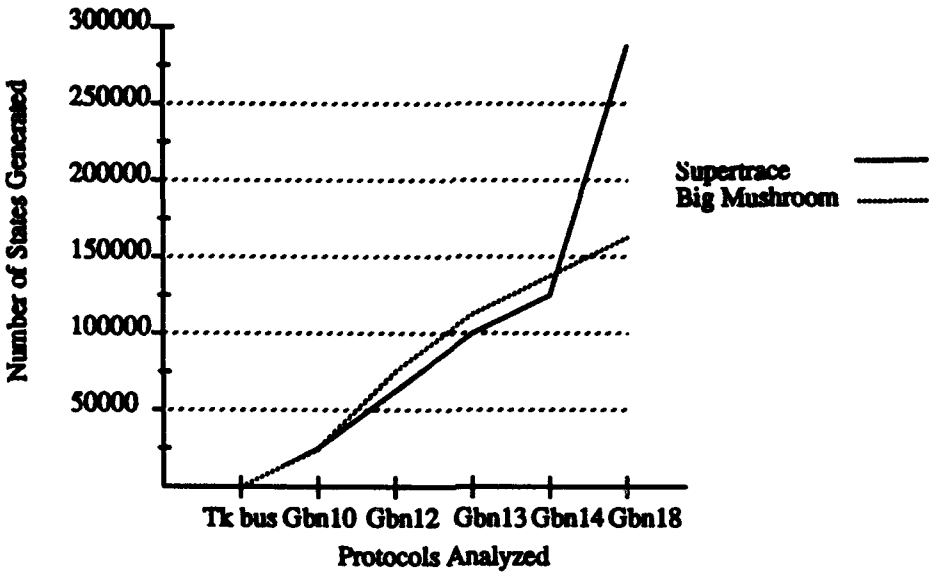

Figure 56 : The Analysis Results of Supertrace and Big Mushroom

The main achievement of Supertrace can be realized when the memory capacity is insufficient to allow an exhaustive analysis. In the analysis of Go-Back-N protocol with a window size 18, Big Mushroom cannot complete the analysis due to insufficient memory. The number of states analyzed with Big Mushroom is 161431 and the number of states analyzed with Supertrace is 290,980. Since we do not know the total number of global states in this protocol, we can not estimate the exact coverage established by Supertrace but we do know that it analyzed 290980 - 161431 =129549 extra states which is 80% more than the number of states generated and analyzed by Big Mushroom. A similar result is established for protocols specified with CFSM model. The analysis of Lap-B protocol with I and RR frames can not be completed by Simple mushroom program. The number of states analyzed is 153565. The same specification analyzed with Supertrace algorithm, and generated 300456 states which is 95% more than the number of states generated by Simple Mushroom.

The results shows that Supertrace algorithm approximates an exhaustive search analysis for smaller protocols and gradually changes into a controlled partial search method for larger protocols. The Supertrace algorithm cannot guarantee 100% coverage due to possibility of unresolved hash conflicts for small protocols. As a partial search technique (for larger protocols) it is far superior to the exhaustive search technique.

The analysis of protocols specified in CFSM model was conducted on a computer with 64 Mbyte memory, the analysis of protocols specified in SCM model was conducted on a computer

with 48 Mbyte memory. The overall improvement of supertrace algorithm is based on these available memory values. The number of states generated can be increased as the amount of the available memory increases. The supertrace algorithm uses a simple hash table for keeping track of the generated global states. Instead of keeping previously generated states in the hash table, a hash value is calculated and corresponding value in the hash table is set. Each state is checked against the hash table values to determine if it was previously generated.

The number of states analyzed and the coverage can be significantly improved by increasing the hash table size in the main program.The supertrace algorithm is also more efficient in speed than the exhaustive search method, since time spent in checking hash table is constant (O(1)). The total processing time difference between these two methods increases as the number of global states increases.

The number of states analyzed is usually very large and it is hard to locate faults by manually searching the output text file. An improvement would be to store the reachability analysis results in the form of a data base.A query language that allows the user to easily analyze the results of the analysis is suggested in [AGGA87].

The data structures can be simplified to allow more efficient utilization of memory so the user can analyze a larger number of states and obtain a more accurate analysis.

Finally, the mushroom with supertrace is a tool which will greatly improve the analysis of large protocols specified by the SCM and CFSM models which cannot be analyzed with exhaustive search methods.

B. TESTGEN Program

In the second part of this thesis a software tool called "TESTGEN" was introduced which automatically produces a sequence of conformance test for protocols specified by the SCM protocol model. The purpose is to conduct conformance testing on implementations. The TESTGEN program checks key control points in the protocol and informs the user if it detects a possible error.

The TESTGEN program takes as input a protocol specified formally as two separate text files, one containing the finite state machine part, the other containing the predicate-action table and variables. It outputs test sequences beginning from the initial s , finding all transition sequences, excluding continuous cycles, and generates tests for every transition on the path back to the initia state, so long as there is such a path (when there is no path back user is warned).

The main achievement of the TESTGEN program is its applicability to protocols specified formally with the SCM model which make it possible for implementors and buyers/users of protocol implementations to automatically generate a set of tests, which ideally determine if the

protocol implementation meets its specification. It was used to generate test sequences for the FDDI protocol in Chapter V and CSMA/CD protocol in Chapter III. It produced the same test sequence generated for the CSMA/CD protocol in [MILL90]. The automation of the test sequence generation procedure TESTGEN expanded the applicability of the procedure to larger and more complex protocols.

A second, broader purpose of this work has been to unify the fields of protocol specification, testing and verification under a single protocol model, *systems of communicating machines*. As earlier work [BULB93] has automated the verification process (to some degree), we now have tools for specification, verification and testing in this protocol model.

The TESTGEN programs generates a test sequence based on the specification of the protocol and a conformance test originated on these test sequences. It verifies that a given implementation realizes all functions of the original specification, over the range of parameter values. If the *implementation under test* (IUT) passes these tests, it is capable of reproducing the behavior formal specification. We do not know if IUT will handle erroneous inputs in a manner consistent with the original specification. Because conformance test sequence is used to test the presence of desirable behavior, not the absence of undesirable behavior.

A further study on this issue might be the generation of a simulator consistent with the specified protocol such that the expected output values can be calculated quickly. Each step in the transition sequence could also be tested and verified easily. The success of this method will depend on the correctness of the simulator program.

The TESTGEN program is originated from the procedure created in [LUND90A]. Further research in this area might be to improve of the procedure itself and determine what assumptions are made concerning the IUT.

The TESTGEN program does not guarantee detection of all the errors in the protocol. It does represent an attempt to exercise all parts of IUT and provides some assurance that the implementation meets its purpose without obvious or easily detected errors.

APPENDIX A (LAP-B Protocol Information Transfer Phase)

Analysis Results (I Frames Only)

REACHABILITY ANALYSIS of : fad.fsm
SPECIFICATION

Machine 1 State Transitions			
From	To	other machine	Transition
1	1	3	r A0
1	2	3	s D0
2	2	3	r A0
2	3	3	s D1
2	4	3	r A1
3	3	3	r A0
3	5	3	r A1
3	7	3	r A2
4	4	3	r A1
4	5	3	s D1
5	5	3	r A1
5	7	3	r A2
5	6	3	s D2
6	6	3	r A1
6	1	3	r A0
6	8	3	r A2
7	7	3	r A2
7	8	3	s D2
8	8	3	r A2
8	1	3	r A0
8	9	3	s D0
9	9	3	r A2
9	2	3	r A0
9	4	3	r A1

Machine 2 State Transitions			
From	To	other machine	Transition
1	4	3	r ENQ
1	2	3	r D0
2	5	3	r ENQ
2	3	3	r D1
3	6	3	r ENQ
3	1	3	r D2
4	1	3	s A0
5	2	3	s A1
6	3	3	s A2

.
.
.

Machine 6 State Transitions			
From	To	other machine	Transition
1	4	4	r ENQ
1	2	4	r D0
2	5	4	r ENQ
2	3	4	r D1
3	6	4	r ENQ
3	1	4	r D2
4	1	4	s A0
5	2	4	s A1
6	3	4	s A2

REACHABILITY GRAPH

```
1 [ 1,E,E,E,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E]
     -D0  3 [2,E,D0 ,E,E,E,1,E,E,E,E,E,E,1,E,E,E,E,E,E,1,E,E,E,E,E,E,1,E,E,E,E,E,E,1,E,E,E,E,E,E] 2
     -D0  4  [1,E,E,E,E,E,E,1,E,E,E,E,E,E,1,E,E,E,E,E,E,1,E,E,E,E,E,E,2,E,E,E,D0,E,1,E,E,E,E,E,E] 3
2 [ 2,E,D0 ,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E]
     -D1 3 [3,E,D0D1,E,E,E,1,E,E,E,E,E,E,1,E,E,E,E,E,E,1,E,E,E,E,E,E,1,E,E,E,E,E,E,1,E,E,E,E,E,E] 4
    +D0 1 [2,E,E,E,E,E,E,1,E,E,E,E,E,E, 2,E,E,E,E,E,E,1,E,E,E,E,E,E,1,E,E,E,E,E,E,1,E,E,E,E,E,E]   5
     -D0  4 [2,E,D0,E,E,E,1,E,E,E,E,E,E,1,E,E,E,E,E,E,1,E,E,E,E,E,E,2,E,E,E,D0,E,1,E,E,E,E,E,E] 6
3 [ 1,E,E,E,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E, 2,E,E,E,D0 ,E, 1,E,E,E,E,E,E]
     -D0  3  [ 2,E,D0 ,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E, 2,E,E,E,D0 ,E, 1,E,E,E,E,E,E]  0
     +D0  5  [ 1,E,E,E,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E, 2,E,E,E,E,E,E, 2,E,E,E,E,E,E, 1,E,E,E,E,E,E]   7
```

```
        -D1  4  [ 1,E,E,E,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E, 3,E,E,E,DO D1 ,E, 1,E,E,E,E,E,E] 8
    4 [ 3,E,DO D1 ,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E]
        +DO  1  [ 3,E,D1 ,E,E,E, 1,E,E,E,E,E,E, 2,E,E,E,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E]   9
        -DO  4 [ 3,E,DO D1 ,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E,1,E,E,E,E,E,E, 2,E,E,E,DO,E,1,E,E,E,E,E,E] 10
    5 [ 2,E,E,E,E,E,E, 1,E,E,E,E,E,E, 2,E,E,E,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E]
        -D1  3  [ 3,E,D1 ,E,E,E, 1,E,E,E,E,E,E, 2,E,E,E,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E]   0
        -ENQ 2  [ 2,E,E,E,E,E,E, 1,E,E,E,E,E,E, 0,E,ENQ,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E]  11
        -DO  4  [ 2,E,E,E,E,E,E, 1,E,E,E,E,E,E, 2,E,E,E,E,E,E, 1,E,E,E,E,E,E, 2,E,E,E,DO ,E, 1,E,E,E,E,E,E]  12
                                                        .
                                                        .
                                                        .
16917 [ 3,E,E,E,E,E,E, 3,E,E,E,E,E,E, 1,E,E,E,E,E,E, 1,E,E,E,E,E,E,3,E,E,E,E,E,E,3,E,E,E,E,E,E]
 **********DEADLOCK condition**********
16818 [ 6,E,E,E,E,E,E, 2,E,E,E,E,E,E, 1,E,F,I21,E,E, 9,E,E,E,E,E,E, 6,E,E,E,D2 ,E, 6,E,E,E,E,E,E]
        -A2  4  [6,E,E,E,E,E,E,2,E,E,E,E,E,E,1,E,E,I21,E,E,9,E,E,E,E,E,E,6,E,E,E,D2,E,3,E,E,E,A2,E]17498
                                                        .
                                                        .
                                                        .
69102...
```

The result of Lap-B Protocol analysis (I frames only)

```
SUMMARY OF REACHABILITY ANALYSIS (ANALYSIS COMPLETED)
-----------------------------------------------------
Total number of states generated : 69102
Number of states analyzed : 69102
number of deadlocks : 1
number of unspecified receptions : 0
maximum message queue size :  6
channel overflow :NONE

                        UNEXECUTED TRANSITIONS
                            *****NONE*****
```

Lap-B Protocol FSM Text File (I and RR frames)

```
start
number_of_machines 6
machine 1
state 1
trans +A0 1 3
trans -D0 2 3
state 2
trans +A0 2 3
trans -D1 3 3
trans +A1 4 3
state 3
trans +A0 3 3
trans +A1 5 3
trans +A2 7 3
state 4
trans +A1 4 3
trans -D1 5 3
state 5
trans +A1 5 3
trans +A2 7 3
trans -D2 6 3
state 6
trans +A1 6 3
trans +A0 1 3
trans +A2 8 3
state 7
trans +A2 7 3
trans -D2 8 3
state 8
trans +A2 8 3
trans +A0 1 3
trans -D0 9 3
state 9
trans +A2 9 3
trans +A0 2 3
trans +A1 4 3
machine 2
state 1
trans +ENQ 10 3
trans +D0 2 3
state 2
trans +ENQ 13 3
trans +D1 3 3
trans -AC1 4 3
state 3
trans +ENQ 14 3
```

```
trans -AC2 7 3
state 4
trans +D1 5 3
trans +ENQ 11 3
state 5
trans +ENQ 14 3
trans -AC2 7 3
trans +D2 6 3
state 6
trans -AC0 1 3
trans +ENQ 15 3
state 7
trans +ENQ 12 3
trans +D2 8 3
state 8
trans -AC0 1 3
trans +ENQ 15 3
trans +D0 9 3
state 9
trans -AC1 4 3
trans +ENQ 13 3
state 10
trans -A0 1 3
state 11
trans -A1 4 3
state 12
trans -A2 7 3
state 13
trans -A1 4 3
state 14
trans -A2 7 3
state 15
trans -A0 1 3
machine 3
state 1
trans +D0 2 1
trans +D1 3 1
trans +D2 4 1
trans +I00 20 4
trans +I10 21 4
trans +I20 22 4
trans +I01 23 4
trans +I11 24 4
trans +I21 25 4
trans +I02 26 4
trans +I12 27 4
trans +I22 28 4
trans +AC0 5 2
trans +AC1 6 2
trans +AC2 7 2
trans +RR0 29 4
trans +RR1 30 4
trans +RR2 31 4
state 2
trans -ENQ 8 2
state 3
trans -ENQ 9 2
state 4
trans -ENQ 10 2
state 5
trans -RR0 1 4
state 6
trans -RR1 1 4
state 7
trans -RR2 1 4
state 8
trans +A0 11 2
trans +A1 12 2
trans +A2 13 2
trans +AC0 8 2
trans +AC1 8 2
trans +AC2 8 2
state 9
trans +A0 14 2
trans +A1 15 2
trans +A2 16 2
trans +AC0 9 2
trans +AC1 9 2
trans +AC2 9 2
state 10
trans +A0 17 2
trans +A1 18 2
trans +A2 19 2
trans +AC0 10 2
trans +AC1 10 2
trans +AC2 10 2
state 11
```

```
trans -I00 1 4
state 12
trans -I01 1 4
state 13
trans -I02 1 4
state 14
trans -I10 1 4
state 15
trans -I11 1 4
state 16
trans -I12 1 4
state 17
trans -I20 1 4
state 18
trans -I21 1 4
state 19
trans -I22 1 4
state 20
trans -D0 29 2
state 21
trans -D1 29 2
state 22
trans -D2 29 2
state 23
trans -D0 30 2
state 24
trans -D1 30 2
state 25
trans -D2 30 2
state 26
trans -D0 31 2
state 27
trans -D1 31 2
state 28
trans -D2 31 2
state 29
trans -A0 1 1
state 30
trans -A1 1 1
state 31
trans -A2 1 1
machine 4
state 1
trans +D0 2 5
trans +D1 3 5
trans +D2 4 5
trans +I00 20 3
trans +I10 21 3
trans +I20 22 3
trans +I01 23 3
trans +I11 24 3
trans +I21 25 3
trans +I02 26 3
trans +I12 27 3
trans +I22 28 3
trans +AC0 5 6
trans +AC1 6 6
trans +AC2 7 6
trans +RR0 29 3
trans +RR1 30 3
trans +RR2 31 3
state 2
trans -ENQ 8 6
state 3
trans -ENQ 9 6
state 4
trans -ENQ 10 6
state 5
trans -RR0 1 3
state 6
trans -RR1 1 3
state 7
trans -RR2 1 3
state 8
trans +A0 11 6
trans +A1 12 6
trans +A2 13 6
trans +AC0 8 6
trans +AC1 8 6
trans +AC2 8 6
state 9
trans +A0 14 6
trans +A1 15 6
trans +A2 16 6
trans +AC0 9 6
trans +AC1 9 6
trans +AC2 9 6
```

```
state 10
trans +A0 17 6
trans +A1 18 6
trans +A2 19 6
trans +AC0 10 6
trans +AC1 10 6
trans +AC2 10 6
state 11
trans -I00 1 3
state 12
trans -I01 1 3
state 13
trans -I02 1 3
state 14
trans -I10 1 3
state 15
trans -I11 1 3
state 16
trans -I12 1 3
state 17
trans -I20 1 3
state 18
trans -I21 1 3
state 19
trans -I22 1 3
state 20
trans -D0 29 6
state 21
trans -D1 29 6
state 22
trans -D2 29 6
state 23
trans -D0 30 6
state 24
trans -D1 30 6
state 25
trans -D2 30 6
state 26
trans -D0 31 6
state 27
trans -D1 31 6
state 28
trans -D2 31 6
state 29
trans -A0 1 5
state 30
trans -A1 1 5
state 31
trans -A2 1 5
machine 5
state 1
trans +A0 1 4
trans -D0 2 4
state 2
trans +A0 2 4
trans -D1 3 4
trans +A1 4 4
state 3
trans +A0 3 4
trans +A1 5 4
trans +A2 7 4
state 4
trans +A1 4 4
trans -D1 5 4
state 5
trans +A1 5 4
trans +A2 7 4
trans -D2 6 4
state 6
trans +A1 6 4
trans +A0 1 4
trans +A2 8 4
state 7
trans +A2 7 4
trans -D2 8 4
state 8
trans +A2 8 4
trans +A0 1 4
trans -D0 9 4
state 9
trans +A2 9 4
trans +A0 2 4
trans +A1 4 4
machine 6
state 1
trans +ENQ 10 4
trans +D0 2 4
```

```
state 2
trans +ENQ 13 4
trans +D1 3 4
trans -AC1 4 4
state 3
trans +ENQ 14 4
trans -AC2 7 4
state 4
trans +D1 5 4
trans +ENQ 11 4
state 5
trans +ENQ 14 4
trans -AC2 7 4
trans +D2 6 4
state 6
trans -AC0 1 4
trans +ENQ 15 4
state 7
trans +ENQ 12 4
trans +D2 8 4
state 8
trans -AC0 1 4
trans +ENQ 15 4
trans +D0 9 4
state 9
trans -AC1 4 4
trans +ENQ 13 4
state 10
trans -A0 1 4
state 11
trans -A1 4 4
state 12
trans -A2 7 4
state 13
trans -A1 4 4
state 14
trans -A2 7 4
state 15
trans -A0 1 4
initial_state 1 1 1 1 1 1
finish
```

The result of Lap-B Protocol analysis (I and RR frames)

```
SUMMARY OF REACHABILITY ANALYSIS (ANALYSIS COMPLETED)
-----------------------------------------------------
Total number of states generated : 320457
Number of states analyzed : 300456
number of deadlocks : 0
number of unspecified receptions : 0
maximum message queue size :  5
channel overflow :NONE

                    UNEXECUTED TRANSITIONS
                        *****NONE*****
```

APPENDIX B (GO BACK N PROTOCOL)

Variable Definitions (Window Size 10)

```
with TEXT_IO; use TEXT_IO;
package definitions  is
   num_of_machines : constant := 2;
   type scm_transition_type is
(snd_data,rcv_data,rcv_ack0,rcv_ack1,rcv_ack2,rcv_ack3,rcv_ack4,rcv_ack5,rcv_ack6,rcv_ack7
,rcv_ack8,rcv_ack9,snd_ack,unused);
   type buffer_type is (d0,d1,d2,d3,d4,d5,d6,d7,d8,d9,E);
   package buff_enum_io is new enumeration_IO(buffer_type);
   use buff_enum_io;
   type buffer_array_type is array(1..10) of buffer_type;
   type seq_array_type is array(1..10) of integer range -1..10;

    type machine1_state_type is
       record
         Sdata: buffer_array_type:= (d0,d1,d2,d3,d4,d5,d6,d7,d8,d9);
         seq  : integer range 0..10 := 0;
         i    : integer range 1..10 := 1;
      end record;

    type machine2_state_type is
       record
         Rdata : buffer_type:= E;
         exp   : integer range 0..10 := 0;
         j     : integer range 1..10 := 1;
     end record;
     type dummy_type is range 1..255;
     type machine3_state_type is
       record
         dummy: dummy_type;
     end record;

                                          •

                                          •

                                          •

     type machine8_state_type is
        record
         dummy: dummy_type;
        end record;

     type global_variable_type is
        record
          DATA : buffer_array_type := (E,E,E,E,E,E,E,E,E,E);
          SEQ  : seq_array_type    := (-1,-1,-1,-1,-1,-1,-1,-1,-1,-1);
          ACK  : integer range  -1..10 := -1;
        end record;
 end definitions;
```

Predicate-Action Table (Window Size 10)

```
separate(main)
procedure Analyze_Predicates_Machine1(local : machine1_state_type;
                                      global : global_variable_type;
                                      s : natural;
                                      w : in out transition_stack_package.stack)  is
```

```
 temp1   : integer := GLOBAL.ACK + 0;
 temp2   : integer := (GLOBAL.ACK + 1) mod 11;
 temp3   : integer := (GLOBAL.ACK + 2) mod 11;
 temp4   : integer := (GLOBAL.ACK + 3) mod 11;
 temp5   : integer := (GLOBAL.ACK + 4) mod 11;
 temp6   : integer := (GLOBAL.ACK + 5) mod 11;
 temp7   : integer := (GLOBAL.ACK + 6) mod 11;
 temp8   : integer := (GLOBAL.ACK + 7) mod 11;
 temp9   : integer := (GLOBAL.ACK + 8) mod 11;
 temp10  : integer := (GLOBAL.ACK + 9) mod 11;
begin
 case s is
   when 0 =>
     if ( (GLOBAL.DATA(local.i) = E ) and (GLOBAL.SEQ(local.i) = -1) ) then
       Push(w,snd_data);
     end if;
   when 1 =>
     if ( (GLOBAL.DATA(local.i) = E ) and (GLOBAL.SEQ(local.i) = -1) ) then
       Push(w,snd_data);
     end if;
     if ( (temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
       Push(w,rcv_ack0);
     end if;
   when 2 =>
     if ( (GLOBAL.DATA(local.i) = E ) and (GLOBAL.SEQ(local.i) = -1) ) then
       Push(w,snd_data);
     end if;
     if ( (temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
       Push(w,rcv_ack0);
     end if;
     if ( (temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
       Push(w,rcv_ack1);
     end if;
   when 3 =>
     if ( (GLOBAL.DATA(local.i) = E ) and (GLOBAL.SEQ(local.i) = -1) ) then
       Push(w,snd_data);
     end if;
     if ( (temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
       Push(w,rcv_ack0);
     end if;
     if ( (temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
       Push(w,rcv_ack1);
     end if;
     if ( (temp3 = local.seq) and (GLOBAL.ACK /= -1)) then
       Push(w,rcv_ack2);
     end if;
   when 4 =>
     if ( (GLOBAL.DATA(local.i) = E ) and (GLOBAL.SEQ(local.i) = -1) ) then
       Push(w,snd_data);
     end if;
     if ( (temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
       Push(w,rcv_ack0);
     end if;
     if ( (temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
       Push(w,rcv_ack1);
     end if;
     if ( (temp3 = local.seq) and (GLOBAL.ACK /= -1)) then
       Push(w,rcv_ack2);
     end if;
     if ( (temp4 = local.seq) and (GLOBAL.ACK /= -1)) then
       Push(w,rcv_ack3);
```

```
  end if;
when 5 =>
  if ( (GLOBAL.DATA(local.i) = E ) and (GLOBAL.SEQ(local.i) = -1) ) then
    Push(w,snd_data);
  end if;
  if ( (temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
    Push(w,rcv_ack0);
  end if;
  if ( (temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
    Push(w,rcv_ack1);
  end if;
  if ( (temp3 = local.seq) and (GLOBAL.ACK /= -1)) then
    Push(w,rcv_ack2);
  end if;
  if ( (temp4 = local.seq) and (GLOBAL.ACK /= -1)) then
    Push(w,rcv_ack3);
  end if;
  if ( (temp5 = local.seq) and (GLOBAL.ACK /= -1)) then
    Push(w,rcv_ack4);
  end if;
when 6 =>
  if ( (GLOBAL.DATA(local.i) = E ) and (GLOBAL.SEQ(local.i) = -1) ) then
    Push(w,snd_data);
  end if;
  if ( (temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
    Push(w,rcv_ack0);
  end if;
  if ( (temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
    Push(w,rcv_ack1);
  end if;
  if ( (temp3 = local.seq) and (GLOBAL.ACK /= -1)) then
    Push(w,rcv_ack2);
  end if;
  if ( (temp4 = local.seq) and (GLOBAL.ACK /= -1)) then
    Push(w,rcv_ack3);
  end if;
  if ( (temp5 = local.seq) and (GLOBAL.ACK /= -1)) then
    Push(w,rcv_ack4);
  end if;
  if ( (temp6 = local.seq) and (GLOBAL.ACK /= -1)) then
    Push(w,rcv_ack5);
  end if;
when 7 =>
  if ( (GLOBAL.DATA(local.i) = E ) and (GLOBAL.SEQ(local.i) = -1) ) then
    Push(w,snd_data);
  end if;
  if ( (temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
    Push(w,rcv_ack0);
  end if;
  if ( (temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
    Push(w,rcv_ack1);
  end if;
  if ( (temp3 = local.seq) and (GLOBAL.ACK /= -1)) then
    Push(w,rcv_ack2);
  end if;
  if ( (temp4 = local.seq) and (GLOBAL.ACK /= -1)) then
    Push(w,rcv_ack3);
  end if;
  if ( (temp5 = local.seq) and (GLOBAL.ACK /= -1)) then
    Push(w,rcv_ack4);
  end if;
```

```
    if ( (temp6 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack5);
    end if;
    if ( (temp7 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack6);
    end if;
  when 8 =>
    if ( (GLOBAL.DATA(local.i) = E ) and (GLOBAL.SEQ(local.i) = -1) ) then
      Push(w,snd_data);
    end if;
    if ( (temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack0);
    end if;
    if ( (temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack1);
    end if;
    if ( (temp3 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack2);
    end if;
    if ( (temp4 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack3);
    end if;
    if ( (temp5 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack4);
    end if;
    if ( (temp6 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack5);
    end if;
    if ( (temp7 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack6);
    end if;
    if ( (temp8 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack7);
    end if;
  when 9 =>
    if ( (GLOBAL.DATA(local.i) = E ) and (GLOBAL.SEQ(local.i) = -1) ) then
      Push(w,snd_data);
    end if;
    if ( (temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack0);
    end if;
    if ( (temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack1);
    end if;
    if ( (temp3 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack2);
    end if;
    if ( (temp4 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack3);
    end if;
    if ( (temp5 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack4);
    end if;
    if ( (temp6 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack5);
    end if;
    if ( (temp7 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack6);
    end if;
    if ( (temp8 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack7);
```

```
      end if;
      if ( (temp9 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w,rcv_ack8);
      end if;
      if ( (temp10 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push (w,rcv_ack9);
      end if;
    when 10 =>
      if ( (temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w,rcv_ack0);
      end if;
      if ( (temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w,rcv_ack1);
      end if;
      if ( (temp3 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w,rcv_ack2);
      end if;
      if ( (temp4 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w,rcv_ack3);
      end if;
      if ( (temp5 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w,rcv_ack4);
      end if;
      if ( (temp6 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w,rcv_ack5);
      end if;
      if ( (temp7 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w,rcv_ack6);
      end if;
      if ( (temp8 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w,rcv_ack7);
      end if;
      if ( (temp9 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w,rcv_ack8);
      end if;
      if ( (temp10 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w,rcv_ack9);
      end if;
    when others =>
      null;
  end case;
end Analyze_Predicates_Machine1;
--------------------------------------------------------------------------
separate(main)
procedure Analyze_Predicates_Machine2(local : machine2_state_type;
                                      global : global_variable_type;
                                      s : natural;
                                      w : in out transition_stack_package.stack)  is

begin
  case s is
    when 0 =>
      if ((GLOBAL.DATA(local.j)/= E) and (GLOBAL.SEQ(local.j) = local.exp)) then
        push(w,rcv_data);
      end if;
    when 1|2|3|4|5|6|7|8|9 =>
      if (GLOBAL.DATA(local.j)= E) then
        Push(w,snd_ack);
      end if;
      if ((GLOBAL.DATA(local.j)/= E) and (GLOBAL.SEQ(local.j) = local.exp)) then
        Push(w,rcv_data);
```

```
      end if;
    when 10 =>
      if (GLOBAL.DATA(local.j)=E) then
        Push(w,snd_ack);
       end if;
    when others =>
      null;
  end case;
end Analyze_Predicates_Machine2;
-------------------------------------------------------------------------
separate(main)
procedure Analyze_Predicates_Machine3(local : machine3_state_type;
                                      global : global_variable_type;
                                      s : natural;
                                      w : in out transition_stack_package.stack) is

begin
  null;
end Analyze_Predicates_Machine3;
-------------------------------------------------------------------------
                                          •

                                          •

                                          •
separate(main)
procedure Analyze_Predicates_Machine8(local : machine8_state_type;
                                      global : global_variable_type;
                                      s : natural;
                                      w : in out transition_stack_package.stack)  is


begin
  null;

end Analyze_Predicates_Machine8;
-------------------------------------------------------------------------
separate(main)
procedure Action ( in_system_state : in out Gstate_record_type;
                   in_transition : in out scm_transition_type;
                   out_system_state : in out Gstate_record_type) is

begin
  case (in_transition) is
    when snd_data =>
      out_system_state.GLOBAL_VARIABLES.DATA(in_system_state.machine1_state.i):=
      in_system_state.machine1_state.Sdata(in_system_state.machine1_state.i);
      out_system_state.GLOBAL_VARIABLES.SEQ(in_system_state.machine1_state.i):=
                            in_system_state.machine1_state.seq;
      out_system_state.machine1_state.i:= (in_system_state.machine1_state.i mod 10) +1 ;
    out_system_state.machine1_state.seq :=(((in_system_state.machine1_state.seq)+1)mod11);
    when rcv_ack0 |rcv_ack1 |rcv_ack2 |rcv_ack3 |rcv_ack4 |rcv_ack5 |rcv_ack6 |rcv_ack7 |
         rcv_ack8 |rcv_ack9   =>
      out_system_state.GLOBAL_VARIABLES.ACK := -1;
    when snd_ack =>
      out_system_state.GLOBAL_VARIABLES.ACK := in_system_state.machine2_state.exp;
      out_system_state.machine2_state.Rdata := e ;
    when rcv_data =>
      out_system_state.machine2_state.Rdata := in_system_state.GLOBAL_VARIABLES.DATA
              (in_system_state.machine2_state.j);
```

```
      out_system_state.GLOBAL_VARIABLES.DATA(in_system_state.machine2_state.j) := E;
      out_system_state.GLOBAL_VARIABLES.SEQ(in_system_state.machine2_state.j) := -1;
      out_system_state.machine2_state.j := (in_system_state.machine2_state.j mod 10) +1;
    out_system_state.machine2_state.exp:=(((in_system_state.machine2_state.exp)+1)mod 11);
    when others =>
      put("Error in action procedure");
  end case;
end Action;
```

Output Format

```
separate(main)
procedure output_Gtuple(tuple : in out Gstate_record_type) is
begin
  if print_header then
    new_line(2);
    set_col(7);
    put_line("  m1(seq,i,Sdata)   , m2(exp,j,Rdata)   , (DATA,SEQ,ACK) ");
    print_header := false;
  else
    put("  [" & integer'image(tuple.machine_state(1)) );
    put(" , ");
    put(tuple.machine1_state.seq,width => 1);
    put(" , ");
    put(tuple.machine1_state.i,width => 1);
    put(" , ");
    buff_enum_io.put(tuple.machine1_state.Sdata(1), set => upper_case);
    put(" , ");
    put( integer'image(tuple.machine_state(2)) );
    put(" , ");
    put(tuple.machine2_state.exp,width => 1);
    put(" , ");
    put(tuple.machine2_state.j,width => 1);
    put(" , ");
    buff_enum_io.put(tuple.machine2_state.Rdata,set => upper_case);
    for i in 1..10 loop
      put(",");
      buff_enum_io.put(tuple.GLOBAL_VARIABLES.DATA(i),set => upper_case);
      put(",");
      put(tuple.GLOBAL_VARIABLES.SEQ(i), width => 1);
    end loop;
    put(" , ");
    put(tuple.GLOBAL_VARIABLES.ACK, width => 1);
    put(" ]");
  end if;
end output_Gtuple;
```

Global Hash Function (Window Size 10)

```
function  GLOBAL_HASH ( current_gstate : Gstate_record_type) return integer is
   index: integer   :=0;
   sum1,sum2:integer:=0;
   m                : machine_state_array := current_gstate.machine_state;
 begin
   index :=  ( (m(8) *83999) + ( m(7) * 72888) + (m(6) *61997) + (m(5) *5995) +
           -(m(4) * 46571) +(m(3) * 34677) + (m(2) * 21323) + (m(1) *18203) ) ;
   sum1 := buffer_type'pos(current_gstate.machine1_state.Sdata(current_gstate.machine1_state.i));
   sum1:= sum1+(23323*current_gstate.machine1_state.seq+31107*current_gstate.machine1_state.i);
   sum1:= sum1 +(20331*buffer_type'pos(current_gstate.machine2_state.Rdata)+
              (19977*current_gstate.machine2_state.exp+17773*current_gstate.machine2_state.j));
   for i in 1..10 loop
     sum2 := sum2+buffer_type'pos(current_gstate.global_variables.DATA(i))*1112*i+
             current_gstate.global_variables.SEQ(i)*3371*2*i;
   end loop;
   return ((index*5+sum1*7+11*sum2+7231*current_gstate.global_variables.ACK) mod 1545423);
 end GLOBAL_HASH;
```

The result of t Go Back N Protocol analysis(Window size 10)

```
SUMMARY OF REACHABILITY ANALYSIS (ANALYSIS COMPLETED)
-----------------------------------------------
Number of states generated :30632
Number of states analyzed :30632
Number of deadlocks : 0
                              UNEXECUTED TRANSITIONS
                                 *****NONE*****
```

The result of the Go back N Protocol analysis(Window size 12)

```
SUMMARY OF REACHABILITY ANALYSIS (ANALYSIS COMPLETED)
-----------------------------------------------
Number of states generated :66655
Number of states analyzed :66655
Number of deadlocks : 0

                              UNEXECUTED TRANSITIONS
                                 *****NONE*****
```

The result of the Go back N Protocol analysis(Window size 13)

```
SUMMARY OF REACHABILITY ANALYSIS (ANALYSIS COMPLETED)
-----------------------------------------------
Number of states generated :90210
Number of states analyzed :90210
Number of deadlocks : 0

                              UNEXECUTED TRANSITIONS
                                 *****NONE*****
```

The result of the Go back N Protocol analysis(Window size 14)

```
SUMMARY OF REACHABILITY ANALYSIS (ANALYSIS COMPLETED)
-----------------------------------------------
Number of states generated :122880
Number of states analyzed :122880
Number of deadlocks : 0

                              UNEXECUTED TRANSITIONS
                                 *****NONE*****
```

The result of the Go back N Protocol analysis(Window size 18)

```
SUMMARY OF REACHABILITY ANALYSIS (ANALYSIS COMPLETED)
-----------------------------------------------
Number of states generated :290980
Number of states analyzed :290980
Number of deadlocks : 0

                              UNEXECUTED TRANSITIONS
                                 *****NONE*****
```

LIST OF REFERENCES

[LUND91] Lundy, G.M., and Miller, R. E., "Specification and Analysis of a Data Transfer Protocol Using Systems of Communicating Machines" *Distributed Computing*, Springer Verlag, December 1991.

[LUND88] Lundy, G. M. and Miller, R. E., "Specification and Analysis of a General Data Transfer Protocol" Tech Rep GIT-88/12, School of Information and Computer Science, Georgia Institute of technology, Atlanta, Georgia, 1988.

[LUND90A] Lundy, G. M., and Miller, R. E., "Testing Protocol Implementations Based on a Formal Specification" IFIP TC Third International Workshop on Protocol Test Systems, Mclean, Virginia, November 1990.

[PENG91] Peng Wuxu and Puroshothaman, S., "Data Flow Analysis of Communicating Finite State Machines" ACM Transactions on Programing Languages and Systems, Vol. 13, No. 3, July 1991.

[RUDI86] Rudin, H., "An Informal Overview of Formal Protocol Specification" IFIP TC 6th International Conference on Information Network and Data Communication, Ronneby Brunn, Sweden, 11-14 May 1986.

[VUON83] Vuong, S. T., and Cowan, D. D, "Reachability Analysis of Protocols with Fifo Channels", ACM SIGCOMM, University of Texas at Austin, March 8-9 1983.

[GOUD83] Gouda, M. G., "An example for Constructing Communicating Machines by stepwise Refinement", Proc. 3rd IFIP WG6.1 Int. Workshop on Protocol Specification, Testing and Verification, North-Holland Publ., 1983.

[HOLZ91] Holzmann, G. J., *Design and Validation of Computer Protocols*, Prentice Hall Software Series, 1991

[ANSIMIL93] United States, Department of Defence, "Reference Manual for the Ada Programming Language," ANSI/MIL-STD-1815A-1983.

[MILL90] Miller, Raymond E., "Protocol Verification: The First Ten Years, The Next Ten Years; Some Personal Observations," *Protocol Specification, Testing and Verification X*, North Holland,1990.

[CHEN90] Chen, C. H., Lu, C. S., Chen, L. and Wang, J. T., "Syncronizable Protocol Test Generation via the Duplex Technique," IEEE INFOCOM, 1990.

[DAHB90] Dahbura, Anton T., Sabnani, Krishan K. and Uyar, M. Umit, "Algorithmic Generation of Protocol Conformance Tests," AT&T Technical Journal,Vol.69 No. 1, Jan-Feb 1990.

[SIDH88] Sidhu, Deepinder and Leung, Ting-kau, "Fault Coverage of Protocol Test Methods," IEEE 1988.

[LUND93] Lundy, G. M., and Miller, Raymond E., "Analyzing a CSMA/CD Protocol through a Systems of communicating Machines Specification," *IEEE Transactions on Communications*, March 1993.

[BULB93] Bulbul, B. "A Protocol Validator for The SCM and CFSM Models,"M. S. Thesis, Department of Computer Science, Naval Postgraduate School, Monterey, California, 1993

[LUND86] Lundy G. M., "Modeling and Analysis of Data Link Protocols," TN86-499.1, Telecommunications Research Laboratory, GTE Laboratories, 40 Slyvan Road, Waltham, Massachusetts, January 1986.

[CHAR90] Charbonneau, L. J., "Specification and Analysis of The Token Bus Protocol," M.S. Thesis, Department of Computer Science, Naval Postgraduate School, Monterey, California, 1990.

[LUND90B] Lundy, G. M., Akyildiz, I. F., "Specification and Analysis of the FDDI MAC Protocol using systems of communicating machines," *Computer Communications*, Vol. 15, No.5, pp. 286-294, June 1992.

[AGGA87] Aggarwal S., Barbara D., and Meth K. Z., "SPANNER: A Tool for the Specification, Analysis, and Evolution of Protocols," IEEE Transactions on Software Engineering, Vol. SE-13, No. 12, December 1987.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexanderia, VA 22304-6145	2
2.	Dudley Knox Library Code 052 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Chairman, Code 37 CS Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	2
4.	Dr. G. M. Lundy, Code CS/Ln Assistant Professor, Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	2
5.	Dr. Lou Stevens, Code CS/St Associate Professor, Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	1
6.	Raymond E. Miller Department of Computer Science A. V. Williams Bldg. University of Maryland College Park, MD 20742	1
7.	Deniz Kuvvetleri Komutanligi Personel Daire Baskanligi Bakanliklar, Ankara /TURKEY	1
8.	Golcuk Tersanesi Komutanligi Golcuk, Kocaeli/ TURKEY	1
9.	Deniz Harp Okulu Komutanligi 81704 Tuzla, Istanbul/TURKEY	1
10.	Taskizak Tersanesi Komutanligi Kasimpasa, Istanbul/TURKEY	1
11.	LTJG. Cuneyt BASARAN Iskenderoglu Sok. No: 31/2 Huzur Apt. Sisli, Istanbul/ TURKEY	1